

Міністерство освіти і науки України
Харківський національний університет імені В.Н. Каразіна
Навчально-науковий інститут комп'ютерних наук та штучного інтелекту

Рекомендовано до захисту:
протокол засідання кафедри
№ ____ від ____ ____ 2025 р.
Завідувач кафедри _____

(підпис) (прізвище та ініціали)

Пояснювальна записка

до кваліфікаційної роботи бакалавра
на тему «Розробка інтелектуального програмного засобу для адаптивного
конфігурування мікросервісних застосунків»

Захищено на засіданні ЕК № _____
протокол № ____ від ____ ____ 2025 р.
Оцінка ____ / _____
Голова ЕК _____

(підпис) (прізвище та ініціали)

Виконав: студент 4 курсу, групи КС-41
Спеціальності:
122 Комп'ютерні науки
Кімлик Денис Анатолійович 

Керівник: ст. викладач КПСіТ
Зінов'єв Д.В. 
Рецензент: к.ф.-м.н., доцент
Споров О.Є.
(ступень, звання, прізвище та ініціали, підпис)

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Харківський національний університет імені В.Н.Каразіна

Навчально-науковий інститут комп'ютерних наук та штучного інтелекту

Кафедра інтелектуальних програмних систем і технологій

Рівень вищої освіти (освітньо-кваліфікаційний рівень) бакалавр

Спеціальність 122 Комп'ютерні науки

ЗАТВЕРДЖУЮ

В.о. завідувача кафедри ІПСіТ

_____ Олег ОЛЕШКО
підпис ім'я, прізвище

“ _____ ” _____ 2025 року

**З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ РОБОТУ**

Кімлика Дениса Анатолійовича

(прізвище, ім'я, по батькові студента)

1. Тема роботи: «Розробка інтелектуального програмного засобу для адаптивного конфігурування мікросервісних застосунків».

керівник роботи старший викладач КІПСіТ Зінов'єв Дмитро Володимирович
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по університету від “16” квітня 2025 року № 4101-5/962

2. Строк подання студентом роботи 02.06.2025 р.

3. Перелік питань, які потрібно розробити:

ознайомитись з особливостями мікросервісної архітектури розподілених програмних систем, провести огляд існуючих методів і програмних засобів конфігурування МСА, познайомитися і зробити аналіз підходів до конфігурування МСА з використанням методів штучного інтелекту, зокрема методу аналізу прецедентів (CBR), створити програму-полігон з мікросервісною архітектурою для проведення експериментальних досліджень, створити програмний застосунок з використанням CBR методу для конфігурування мікросервісів, провести експериментальне дослідження ефективності використання CBR методу для управління мікросервісами.

4. План роботи

№ з/п	Назви етапів роботи
1	Формулювання мети та повного переліку завдань для виконання кваліфікаційної роботи (КР).
2	Пошук та аналіз інформаційних джерел за темою КР.
3	Знайомство з особливостями мікросервісної архітектури розподілених програмних систем.
4	Огляд методів і програмних застосунків для управління конфігураціями програм з мікросервісною архітектурою.
5	Знайомство з методами і аналіз підходів до конфігурування МСА з використанням методів штучного інтелекту.
6	Знайомство з методом аналізу прецедентів (CBR метод) і програмна реалізація алгоритмів вилучення вдалого прецеденту (Nearest Neighbor, Feature-Based Retrieval, Cluster-Based Retrieval, Indexing and Hashing).
7	Проектування та розробка програми-полігону з мікросервісною архітектурою для проведення експериментальних досліджень.
8	Розробка інтелектуального програмного засобу з використанням CBR методу для конфігурування мікросервісних застосунків.
9	Експериментальне дослідження ефективності використання CBR методу для управління мікросервісами.
10	Підсумок отриманих результатів та оформлення пакету документів для процедури захисту КР в екзаменаційній комісії ННІ КН та ІІІ.

5. Дата видачі завдання 10.12.2024 р.

Студент


(підпис)

Кімлик Д.А.

(ініціали, прізвище)

Керівник роботи


(підпис)

Зінов'єв Д.В.

(ініціали, прізвище)

АНОТАЦІЯ

Звіт про виконання: 110 с., 4 розділи, 58 рис., 4 таблиці, 1 додаток, 29 джерел.

У кваліфікаційній роботі розроблено систему інтелектуального адаптивного управління конфігураціями мікросервісів за допомогою Case-Based Reasoning (CBR) методів.

Об'єкт дослідження: процес адаптивного конфігурування мікросервісних застосунків у розподілених системах в умовах динамічного середовища.

Предмет дослідження: інтелектуальні методи та програмні засоби адаптивного конфігурування мікросервісних застосунків під час їх виконання.

Мета дослідження: розробка інтелектуального програмного засобу, що використовує CBR-підхід для автоматичного налаштування конфігурацій мікросервісів в умовах динамічного середовища, а також проведення експериментального дослідження дієвості цієї системи.

Розроблено тестовий полігон, що симулює мікросервісну архітектуру, та створено інтелектуальну систему АУКМ, яка реалізує п'ять різних CBR-методів для автоматичного конфігурування. Програмна реалізація виконана з використанням технологічного стеку Python, FastAPI, PostgreSQL, Docker, SQLAlchemy та scikit-learn.

Ключові слова: МІКРОСЕРВІС, МІКРОСЕРВІСНА АРХІТЕКТУРА, КОНФІГУРАЦІЯ, АДАПТИВНЕ УПРАВЛІННЯ КОНФІГУРАЦІЯМИ, МЕТРИКА, ШТУЧНИЙ ІНТЕЛЕКТ, АНАЛІЗ ПРЕЦЕДЕНТІВ, МЕТОД АНАЛІЗУ ПРЕЦЕДЕНТІВ, CBR, ТЕСТОВИЙ ПОЛІГОН, FASTAPI.

ANNOTATION

Execution report: 110 pages, 4 chapters, 58 fig., 4 tables, 1 application, 29 sources.

This qualification thesis describes the development of an intelligent system for the adaptive management of microservice configurations using Case-Based Reasoning (CBR) methods.

Object of research: The process of adaptive configuration of microservice applications in distributed systems under a dynamic environment.

Subject of research: Intelligent methods and software tools for the adaptive configuration of microservice applications during their execution.

The aim of the research is the development of an intelligent software tool that utilizes the CBR approach for the automatic adjustment of microservice configurations in a dynamic environment, as well as conducting an experimental study of this system's effectiveness.

A testbed simulating a microservice architecture has been developed, along with an intelligent Adaptive Microservice Configuration Management (AUKM) system that implements five different CBR methods for automatic configuration. The software implementation was carried out using the technology stack of Python, FastAPI, PostgreSQL, Docker, SQLAlchemy, and scikit-learn.

Keywords: MICROSERVICE, MICROSERVICE ARCHITECTURE, CONFIGURATION, ADAPTIVE CONFIGURATION MANAGEMENT, METRIC, ARTIFICIAL INTELLIGENCE, CASE-BASED REASONING, CBR, TESTBED, FASTAPI.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАЧЕНЬ, ТЕРМІНІВ	9
ВСТУП	10
1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧІ	13
1.1 Огляд МСА та її особливостей	13
1.2 Сучасні методи управління конфігураціями в МСА	14
1.3 Проблеми конфігурування в динамічних МСС	16
1.4 Застосування ШІ в управлінні мікросервісами	18
1.5 Аналіз CBR як методу адаптивного конфігурування	20
1.6 Обґрунтування доцільності дослідження	21
1.7 Постановка задачі дослідження	22
2 АНАЛІЗ ІНСТРУМЕНТІВ КОНФІГУРУВАННЯ ПРОГРАМНИХ МІКРОСЕРВІСІВ ТА ВИКОРИСТАННЯ МЕТОДІВ CBR ДЛЯ АУКР	24
2.1 Інструменти конфігурування програмних мікросервісів	24
2.1.1 Аналіз Kubernetes ConfigMaps	24
2.1.2 Аналіз HashiCorp Consul	26
2.1.3 Аналіз Spring Cloud Config	28
2.1.4 Аналіз AWS Config.....	29
2.2 Використання методів CBR для АУКМ.....	31
2.2.1 Загальний принцип роботи CBR у контексті АУКР	31
2.2.2 Аналіз K-Nearest Neighbors.....	32
2.2.3 Аналіз Weighted K-Nearest Neighbors	33
2.2.4 Аналіз Feature-Based Retrieval.....	34
2.2.5 Аналіз Cluster-Based Retrieval	35
2.2.6 Аналіз Indexing and Hashing	35

3	ПРОЄКТУВАННЯ ТЕСТОВОГО ПОЛІГОНУ ТА СИСТЕМИ АУКМ..	37
3.1	Вимоги до тестового полігону та системи АУКМ.....	37
3.1.1	Функціональні вимоги	37
3.1.1	Нефункціональні вимоги	38
3.2	Проектування тестового полігону	39
3.2.1	Опис мікросервісів тестового полігону.....	39
3.2.2	Діаграма прецедентів	41
3.2.3	Структура бази даних.....	42
3.2.4	Архітектура та зв'язки мікросервісів	43
3.2.5	Плановані конфігураційні параметри.....	44
3.3	Проектування системи АУКМ	45
3.3.1	Опис компонентів системи АУКМ.....	45
3.3.2	Діаграма прецедентів для системи АУКМ	47
3.3.3	Структура бази даних системи АУКМ.....	48
3.3.4	Архітектура та зв'язки системи АУКМ	49
3.3.5	Інтеграція АУКМ із тестовим полігоном.....	49
4	ПРОГРАМНА РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ ІНСТРУМЕНТУ АУКМ НА ТЕСТОВОМУ ПОЛІГОНІ.....	52
4.1	Технологічний стек реалізації.....	52
4.2	Реалізація тестового полігону	53
4.2.1	Реалізація auth_service.....	53
4.2.2	Реалізація product_service	55
4.2.3	Реалізація order_service	58
4.2.4	Взаємодія між сервісами.....	60
4.3	Контейнеризація тестового полігону	61

4.4	Структура проєкту тестового полігону	63
4.5	Реалізація системи АУКМ.....	64
4.5.1	Основний функціонал	64
4.5.2	СВР-методи.....	65
4.6	Контейнеризація та структура системи АУКМ.....	66
4.7	Результат розробки полігону та його тестування	68
4.7.1	Результат та тестування auth_service.....	68
4.7.2	Результат та тестування product_service	73
4.7.3	Результат та тестування order_service	74
4.8	Результат розробки системи АУКМ та її тестування	76
4.8.1	Інтерфейс системи АУКМ.....	77
4.8.2	Проведення експериментів із системою АУКМ	84
4.9	Порівняння часу виконання СВР-методів.....	92
ВИСНОВКИ.....		97
ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ		98
ДОДАТОК А.....		101

ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАЧЕНЬ, ТЕРМІНІВ

АУКМ – Адаптивне управління конфігураціями мікросервісів

БД – База даних

МСА – Мікросервісна архітектура

МСС – Мікросервісні системи

УК – Управління конфігураціями

ШІ – Штучний інтелект

API – Application Programming Interface (Інтерфейс програмування додатків)

CBR – Case-Based Reasoning (Метод аналізу прецедентів)

EA – Evolutionary Algorithms (Еволюційні алгоритми)

gRPC – gRPC Remote Procedure Call (Віддалений виклик процедур gRPC)

JSON – JavaScript Object Notation (Формат обміну даними на основі тексту)

KNN – K-Nearest Neighbors (К-найближчих сусідів)

ML – Machine Learning (Машинне навчання)

PA – Predictive Analytics (Передбачувальна аналітика)

QoS – Quality of System (Якість системи)

REST – Representational State Transfer (Стиль архітектури для веб-додатків)

RL – Reinforcement Learning (Навчання з підкріпленням)

ВСТУП

Мікросервісна архітектура стала одним із ключових підходів до розробки сучасних розподілених програмних систем. На відміну від монолітних рішень, вона дозволяє розбити великий складний додаток на невеликі, незалежні сервіси, кожен із яких можна розгортати, оновлювати та масштабувати окремо [1]. Завдяки цьому багато технологічних компаній, зокрема Netflix, Amazon і Google, змогли досягнути високої гнучкості та швидкої адаптації своїх сервісів до змін у бізнес-вимогах. Проте поряд із перевагами приходять і суттєві виклики, зокрема у сфері конфігурування мікросервісів та забезпечення узгодженої роботи всієї системи.

Сучасні платформи для управління конфігураціями, такі як Kubernetes ConfigMaps, Consul і Spring Cloud Config, надають можливість централізовано зберігати й поширювати налаштування серед сервісів. Однак у більшості випадків ці підходи базуються на заздалегідь визначених шаблонах, статичних правилах або ж припускають ручне втручання адміністратора у разі зміни умов роботи [1]. У випадках динамічних змін зовнішніх умов у роботі системи, наприклад навантажень, коли кількість запитів до сервісів може різко зростати чи падати, використання лише статичних конфігурацій призводить до неефективного використання ресурсів, зниження продуктивності та, в окремих випадках, до перевантажень чи простоїв окремих компонентів.

Основною проблемою, з якою стикаються інженери й адміністратори мікросервісних систем, є відсутність механізмів динамічного конфігурування, які могли б автоматично реагувати на зміни в середовищі. Існуючі рішення, такі як шаблони конфігурацій або ручне налаштування, часто не враховують історичних даних про поведінку системи під навантаженням і не здатні передбачити ситуації збудливості (наприклад, раптовий стрибок трафіку чи збій мережевого з'єднання). В результаті адміністратор змушений реагувати руками, що уповільнює процес відновлення нормальної роботи та призводить до використання зайвих ресурсів.

Одним із сучасних і перспективних напрямів подолання цих обмежень є застосування методів штучного інтелекту (ШІ) для аналізу попереднього досвіду роботи системи і прийняття оптимальних рішень щодо змін в її конфігурації в реальному часі. Зокрема, метод аналізу прецедентів дозволяє зберігати в базі випадки (тобто ті самі прецеденти), коли певна комбінація параметрів конфігурації давала очікувані позитивні результати. Коли виникає нова ситуація, система порівнює її ознаки з уже наявними прецедентами й обирає найбільш релевантні рішення, адаптувавши їх до поточних умов [2]. Такий підхід вже широко використовується, наприклад, у медичних діагностичних системах та рекомендованих алгоритмах.

На даний час питання, що пов'язані з використанням методів ШІ для управління конфігураціями програмного забезпечення з мікросервісною архітектурою ще недостатньо вивчені. Тому аналіз і тестування існуючих методів ШІ для управління конфігураціями програмних застосунків на базі мікросервісної архітектури є важливими та актуальними.

Об'єктом дослідження цієї роботи виступає процес адаптивного конфігурування мікросервісних застосунків у розподілених системах в умовах динамічного середовища.

Предметом дослідження є інтелектуальні методи та програмні засоби адаптивного конфігурування мікросервісних застосунків під час їх виконання.

Метою роботи є розробка інтелектуального програмного засобу, що використовує CBR-підхід для автоматичного налаштування конфігурацій мікросервісів в умовах динамічного середовища, а також проведення експериментального дослідження ефективності цієї системи.

Для досягнення поставленої мети передбачено вирішити такі завдання:

1. Ознайомитися з особливостями мікросервісної архітектури, її перевагами й обмеженнями, а також із сучасними засобами управління конфігураціями (Kubernetes, Consul, Spring Cloud Config).

2. Провести огляд існуючих методів адаптивного керування конфігураціями мікросервісів, з акцентом на ШІ-підходи та їхню придатність для динамічних середовищ.
3. Проаналізувати принципи роботи CBR-методології та вибрати алгоритми (Nearest Neighbor, Feature-Based Retrieval, Cluster-Based Retrieval, Indexing and Hashing), які найкраще підходять для адаптивного налаштування параметрів у мікросервісному середовищі.
4. Розробити тестовий «полігон» у вигляді мікросервісного застосунку на базі FastAPI для відпрацювання алгоритмів адаптивного конфігурування.
5. Реалізувати окремий CBR-сервіс (через FastAPI), який взаємодіє з полігоном та пропонує оптимальні конфігурації на основі аналогічних прецедентів.
6. Провести експериментальну оцінку ефективності розробленого рішення.

Необхідність цього дослідження обґрунтовується зростаючими вимогами ринку до гнучких та автоматизованих рішень у сфері DevOps, де традиційні підходи вже не справляються з реаліями динамічних навантажень. Актуальність роботи полягає в тому, що адаптивне інтелектуальне конфігурування може суттєво підвищити стійкість і продуктивність мікросервісних систем, зменшити кількість простоїв і екстрених втручань адміністратора.

Новизна ж роботи полягає у поєднанні CBR-підходу з MCA на базі FastAPI та детальному порівнянні кількох методів вибору прецедентів (Nearest Neighbor, Feature-Based Retrieval, Cluster-Based Retrieval, Indexing and Hashing). Пропонується створити незалежний інтелектуальний мікросервіс, який забезпечує динамічне налаштування параметрів інших сервісів, враховуючи історичний досвід. Це дозволить оперативно реагувати на зміни в робочому середовищі (зростання навантаження, зміни топології в мережі), та оптимізувати використання ресурсів на основі успішних кейсів із минулого.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧІ

1.1 Огляд МСА та її особливостей

Як я вже встиг зазначити раніше у вступі, МСА є сучасним підходом до розробки програмного забезпечення, який передбачає розбиття складної системи на набір невеликих, автономних сервісів, кожен із яких виконує чітко визначену функцію та взаємодіє з іншими через стандартизовані інтерфейси, такі як REST або gRPC [3]. На відміну від монолітних архітектур, де весь функціонал системи реалізується в єдиному застосунку, МСА забезпечує слабку зв'язність між компонентами, що дозволяє незалежне розгортання, масштабування та оновлення кожного мікросервісу [4]. Основними принципами МСА є:

- декомпозиція за бізнес-функціями: кожен мікросервіс відповідає окремій бізнес-можливості (наприклад, обробка платежів, управління користувачами);
- незалежність: мікросервіси можуть розроблятися, розгортатися та масштабуватися окремо;
- стандартизована взаємодія: використовуються протоколи HTTP/HTTPS, повідомлення через черги (наприклад, RabbitMQ) або API;
- автономність даних: кожен мікросервіс має власну базу даних, що знижує залежності [3].

Переваги МСА включають гнучкість у виборі технологій, швидке оновлення окремих компонентів, ізоляцію помилок і покращену масштабованість. Наприклад, у разі зростання навантаження на певний сервіс (скажімо, обробку замовлень в інтернет-магазині) можна масштабувати лише цей сервіс, не зачіпаючи інші [5]. Водночас МСА створює нові виклики, такі як складність координації між сервісами, управління конфігураціями, моніторинг і забезпечення безпеки в розподілених системах. Особливо гостро стоїть проблема конфігурування, оскільки кожен мікросервіс потребує власного набору параметрів (наприклад, підключення до бази даних, ліміти

ресурсів), які можуть динамічно змінюватися залежно від умов експлуатації [4].

Для порівняння монолітних і мікросервісних систем розглянемо їхню структуру (див. рис. 1.1).



Рисунок 1.1 – Порівняння монолітної та мікросервісної архітектур

Для порівняння монолітних і мікросервісних систем розглянемо їхню структуру. У монолітній архітектурі всі компоненти (UI, бізнес-логіка, база даних) тісно пов'язані, що ускладнює внесення змін і масштабування. Натомість МСА дозволяє розподілити компоненти на автономні сервіси, що взаємодіють через API [3]. Ця гнучкість робить МСА популярною в хмарних середовищах, таких як AWS, Azure чи Google Cloud, де динамічне масштабування є ключовою вимогою.

1.2 Сучасні методи управління конфігураціями в МСА

Як було не важко здогадатись з моїх попередніх слів, управління конфігураціями в мікросервісних системах є критично важливим завданням, оскільки кожен мікросервіс потребує власного набору параметрів, які можуть включати адреси серверів, ліміти ресурсів, ключі API чи налаштування безпеки [7]. На даний час вже існує набір інструментів для управління

конфігураціям, які можна поділити на три категорії: централізовані сховища, інструменти оркестрації та платформи автоматизації.

Централізовані сховища конфігурацій.

Такі інструменти, як HashiCorp Consul і Spring Cloud Config, дозволяють зберігати конфігурації в єдиному репозиторії та надавати їх мікросервісам у реальному часі. Consul підтримує динамічне оновлення параметрів і синхронізацію через API, тоді як Spring Cloud Config інтегрується з Git для версіонування конфігурацій [4]. Однак ці інструменти мають обмеження, зокрема залежність від статичних правил і складність обробки великих обсягів даних у реальному часі.

Інструменти оркестрації.

Kubernetes ConfigMaps і Secrets є популярними рішеннями для управління конфігураціями в контейнеризованих середовищах. ConfigMaps дозволяють зберігати некритичні параметри (наприклад, URL сервера), тоді як Secrets призначені для чутливих даних, таких як паролі [9]. Основним недоліком є необхідність ручного оновлення конфігурацій при зміні умов, що ускладнює адаптацію до динамічних навантажень.

Платформи автоматизації.

HashiCorp Terraform і AWS Config забезпечують автоматизацію розгортання та моніторинг конфігурацій. Terraform дозволяє описувати інфраструктуру як код (Infrastructure as Code, IaC), що спрощує масштабування, тоді як AWS Config відстежує зміни в ресурсах і виявляє невідповідності [7]. Проте ці інструменти часто не можуть оперативно реагувати на раптові зміни в системі, такі як сплески трафіку.

В роботі був проведений аналіз існуючих інструментів управління конфігураціями програмних застосунків за наступними критеріями оцінювання: автоматизація; підтримка реального часу; масштабованість; простота інтеграції. Результати аналізу наведені у Таблиці 1.1.

Таблиця 1.1 – Таблиця порівняння інструментів УК

Інструмент	Автоматизація	Підтримка реального часу	Масштабованість	Простота інтеграції
Kubernetes ConfigMaps	Середня (вимагає ручного оновлення)	Низька (затримки в оновленні)	Висока (підходить для контейнерів)	Висока (інтеграція з Kubernetes)
HashiCorp Consul	Висока (динамічне оновлення через API)	Висока (миттєва синхронізація)	Середня (обмеження при великих масштабах)	Середня (вимагає конфігурації)
Spring Cloud Config	Середня (автоматизація через Git)	Середня (залежить від Git)	Висока (підходить для розподілених систем)	Висока (інтеграція з Java)
HashiCorp Terraform	Висока (IaC, автоматизоване розгортання)	Низька (статичне оновлення)	Висока (гнучке масштабування)	Середня (вимагає навчання)
AWS Config	Висока (автоматизован ий моніторинг)	Середня (затримки в обробці)	Висока (хмарна масштабованість)	Висока (інтеграція з AWS)

Як видно з таблиці 1.1, наприклад, Kubernetes ConfigMaps забезпечує високу інтеграцію з контейнерами, але потребує додаткових інструментів для автоматизації, тоді як Consul підтримує динамічні оновлення, але має складну конфігурацію [7]. Обмеженнями всіх цих інструментів є недостатня адаптивність до динамічних умов і залежність від заздалегідь визначених правил, що ускладнює їх використання в системах із непередбачуваними навантаженнями [8]. Це дає можливість зробити висновок, що сучасні методи УК пропонують доволі широкий спектр можливостей, але мають низьку адаптивність і велику складність синхронізації. Тому залишається необхідність у розробці нових підходів до процесу конфігурування мікросервісних застосунків, які могли б ефективно працювати в умовах динамічних змін МСС.

1.3 Проблеми конфігурування в динамічних МСС

Динамічні мікросервісні середовища, які функціонують у хмарних платформах, стикаються з низкою проблем, пов'язаних із конфігуруванням. Однією з ключових проблем є конфігураційний дрейф (configuration drift). Це

невідповідність між бажаним і фактичним станами конфігурацій, яка може виникати через ручні зміни, помилки синхронізації або несвоєчасне оновлення параметрів [7]. Конфігураційний дрейф може призводити до наступних наслідків:

- неоптимальні параметри (наприклад, недостатня кількість реплік сервісу) збільшують затримку та знижують пропускну здатність;
- неправильне налаштування лімітів може призвести до надмірного споживання CPU чи пам'яті;
- невідповідність конфігурацій нормативним вимогам (наприклад, застарілі ключі шифрування) підвищує вразливість системи [5].

На рисунку 1.2 показаний графік, що ілюструє негативні наслідки дрейфу (затримка, споживання ресурсів, доступність) залежно від часу або ступеня дрейфу. [7]

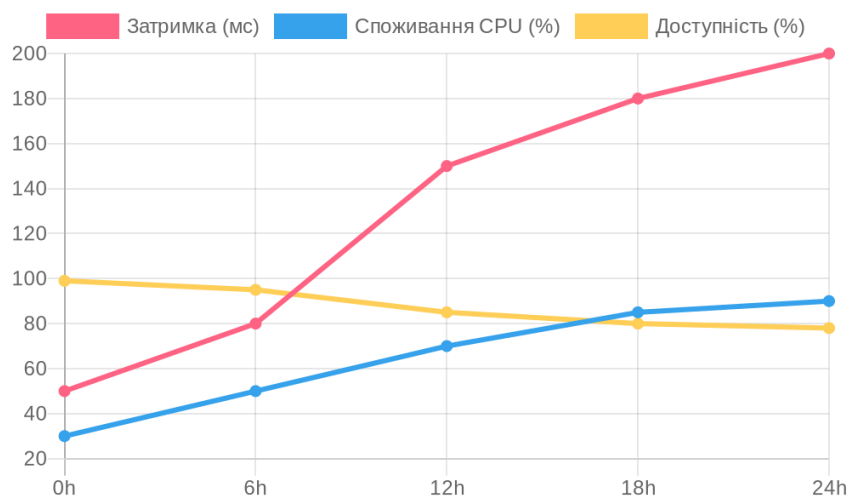


Рисунок 1.2 – Негативні наслідки дрейфу для системи

Іншою проблемою є складність масштабування в умовах раптових сплесків трафіку. Наприклад, у системах електронної комерції під час розпродажів навантаження може зростати в десятки разів, що вимагає швидкої адаптації конфігурацій (збільшення кількості реплік, зміни лімітів ресурсів) [8]. Традиційні інструменти, такі як Kubernetes ConfigMaps чи Terraform, не завжди здатні оперативно реагувати на такі зміни через затримки в оновленні або залежність від ручного втручання [7].

Крім того, у розподілених системах можуть виникати проблеми синхронізації конфігурацій між самими мікросервісами. Наприклад, якщо один сервіс оновив адресу API, а інший продовжує використовувати застарілу, це може призвести до збоїв [4]. Ці виклики підкреслюють необхідність розробки інтелектуальних методів конфігурування, які могли б адаптивно реагувати на зміни в реальному часі та мінімізувати конфігураційний дрейф.

1.4 Застосування ШІ в управлінні мікросервісами

Використання штучного інтелекту для управління мікросервісними системами відкриває нові можливості для автоматизації, оптимізації ресурсів і адаптивного масштабування. Сучасні ШІ-технології, такі як машинне навчання, передбачувальна аналітика, навчання з підкріпленням та еволюційні алгоритми, активно застосовуються для вирішення проблем конфігураційного дрейфу, неефективного використання ресурсів і динамічного навантаження [5].

Машинне навчання може використовуватись для прогнозування навантаження та автоматичного налаштування параметрів мікросервісів. Наприклад, моделі ML можуть аналізувати історичні дані про трафік і рекомендувати оптимальну кількість реплік сервісу [10].

Передбачувальна аналітика дозволяє виявляти аномалії в продуктивності (наприклад, зростання затримки через неоптимальні конфігурації) і пропонувати коригувальні дії до виникнення збоїв [19].

Навчання з підкріпленням застосовується для адаптивного управління ресурсами: агент RL взаємодіє з середовищем, оптимізуючи параметри (наприклад, ліміти CPU) на основі винагороди за зниження затримки чи споживання ресурсів [5].

Еволюційні алгоритми, такі як генетичні алгоритми, використовуються для пошуку оптимальних конфігурацій у складних системах із великою кількістю параметрів, імітуючи природний відбір [11].

При виконанні роботи була проаналізована і побудована схема інтеграції ІІІ-компонентів у МСА, яка наведена на рисунку 1.3.

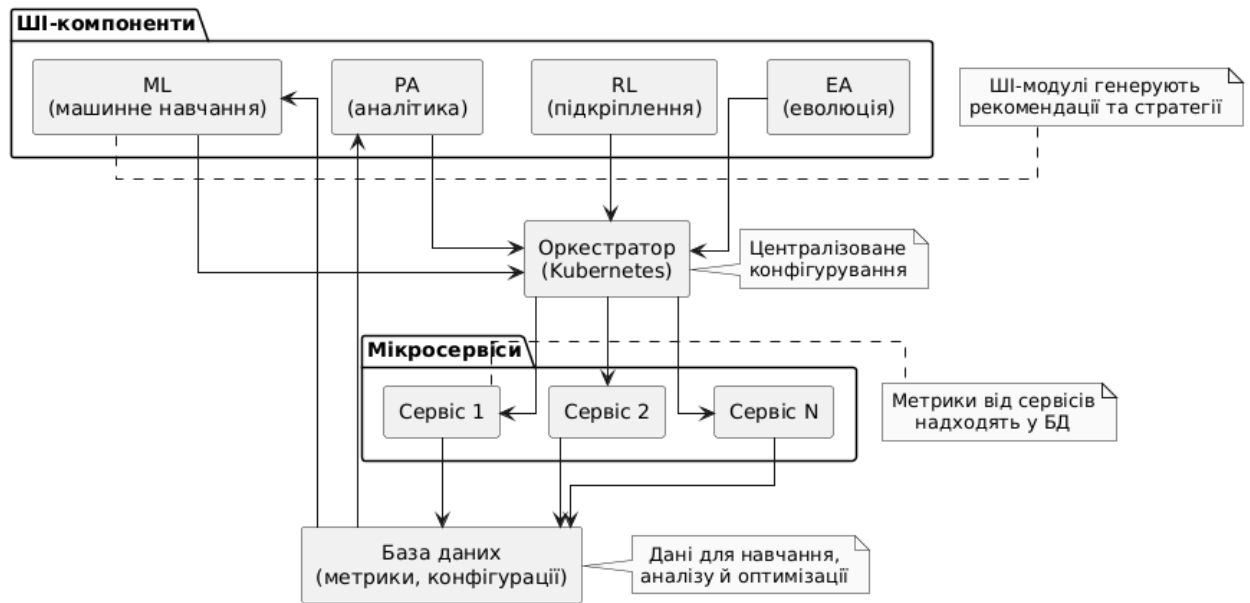


Рисунок 1.3 – Схема інтеграції ІІІ-компонентів у МСА

Схема була розроблена з метою ілюстрації інтеграції ІІІ-технологій із мікросервісною архітектурою через оркестратор Kubernetes, що дозволяє централізовано керувати конфігураціями та метриками. На основі цієї схеми в подальшому планується спроектувати систему, яка включатиме тестовий полігон із мікросервісною архітектурою та окремий модуль для адаптивного управління конфігураціями мікросервісів.

Аналіз літератури щодо можливості використання ІІІ-методів також показав, що вони мають деякі обмеження. По-перше, навчання моделей ML і RL потребує великих обсягів даних, що може бути проблематичним для нових систем із обмеженою історією [10]. По-друге, висока обчислювальна складність алгоритмів (особливо RL і ЕА) вимагає значних ресурсів, що може знизити ефективність у реальному часі [19]. По-третє, складність інтерпретації рішень ІІІ (проблема «чорної скриньки») ускладнює їх використання в критичних системах, де потрібна прозорість [11].

1.5 Аналіз CBR як методу адаптивного конфігурування

CBR є підходом ШІ, який базується на використанні минулого досвіду для вирішення нових проблем. CBR складається з чотирьох етапів: Retrieve (пошук), Reuse (повторне використання), Revise (адаптація) та Retain (збереження) [6]. У контексті мікросервісних систем CBR може застосовуватися для адаптивного конфігурування, використовуючи історичні прецеденти (наприклад, конфігурації, що забезпечили низьку затримку) для вибору оптимальних параметрів у поточних умовах [5]. Ключовим елементом CBR є алгоритми пошуку прецедентів, серед яких:

- Nearest Neighbor: оцінює схожість між поточною ситуацією та прецедентами за метрикою відстані (наприклад, евклідова відстань). Перевага – простота, недолік – низька ефективність при великих базах прецедентів [6];
- Feature-Based Retrieval: фокусується на ключових ознаках (наприклад, навантаження, тип мережі). Перевага – точність, недолік – потреба у визначенні ознак [2];
- Cluster-Based Retrieval: групує прецеденти в кластери для швидшого пошуку. Перевага – масштабованість, недолік – складність кластеризації [5];
- Indexing and Hashing: використовує індекси для прискорення пошуку. Перевага – швидкість, недолік – складність підтримки індексів [2].

Використання цих методів дозволяє адаптувати CBR до специфіки МСА, де конфігурації постійно змінюються через динамічне навантаження та розподілені компоненти. На відміну від статичних підходів, CBR дозволяє системі навчатися на реальних даних, оперативно знаходячи рішення для нових, раніше невідомих станів системи. Таким чином, система не просто реагує на зміни, а й накопичує знання, підвищуючи точність майбутніх конфігураційних рішень. Для ілюстрації цього процесу в контексті МСА на рисунку 1.4 наведена діаграма циклу роботи CBR, що відображає інтеграцію етапів у систему управління конфігураціями.



Рисунок 1.4 – Діаграма циклу роботи CBR для конфігурування мікросервісів

1.6 Обґрунтування доцільності дослідження

Під час аналізу літератури я зрозумів, що сучасні методи управління конфігураціями, такі як Kubernetes ConfigMaps, Consul чи Terraform, мають обмеження, зокрема низьку адаптивність до динамічних умов і залежність від статичних правил (про це вже було сказано вище) [7]. Конфігураційний дрейф, неефективне використання ресурсів і затримки в оновленні параметрів залишаються ключовими проблемами, які знижують продуктивність і підвищують ризики [3]. Застосування ШІ-технологій, таких як RL чи EA, частково вирішує ці проблеми, але їхня складність і потреба у великих обсягах даних обмежують практичне використання [5].

Запропонований підхід, заснований на методології CBR, спрямований на подолання цих обмежень. CBR дозволяє адаптивно конфігурувати мікросервіси, використовуючи історичні прецеденти для швидкого вибору оптимальних параметрів без необхідності складного навчання моделей [6]. Інтеграція CBR із FastAPI забезпечує високу швидкість обробки запитів і

легкість масштабування, що є новизною дослідження. Крім того, порівняння кількох алгоритмів пошуку прецедентів (Nearest Neighbor, Feature-Based Retrieval тощо) дозволить визначити найефективніший підхід саме для МСА [2].

1.7 Постановка задачі дослідження

Метою дослідження є розробка інтелектуального програмного засобу для адаптивного конфігурування мікросервісних систем на основі методології аналізу прецедентів (CBR), реалізованого через фреймворк FastAPI, із подальшою експериментальною оцінкою його ефективності. Задача дослідження полягає в автоматизації процесу вибору оптимальних конфігурацій для мікросервісів у динамічних умовах, що дозволить мінімізувати конфігураційний дрейф, знизити затримку та оптимізувати використання ресурсів [5]. В якості основних компонентів для цієї задачі виступають:

- вхідні дані: історичні прецеденти (набір конфігурацій і відповідних метрик продуктивності), поточні метрики системи (навантаження, затримка, споживання ресурсів);
- вихідні результати: оптимальні конфігурації мікросервісів (наприклад, кількість реплік, ліміти CPU, параметри підключення);
- критерії оцінки: затримка (latency), пропускна здатність (throughput), використання ресурсів (CPU, пам'ять).

В процесі роботи була розроблена блок-схема архітектури запропонованого рішення, яка показана на рисунку 1.5.

Етапи реалізації включають:

1. Розробку бази прецедентів із конфігураціями та метриками.
2. Реалізацію CBR-сервісу через FastAPI для пошуку та адаптації прецедентів.
3. Створення тестового полігону для імітації мікросервісного середовища.

4. Експериментальне порівняння алгоритмів пошуку прецедентів (Nearest Neighbor, Feature-Based Retrieval, Cluster-Based Retrieval, Indexing and Hashing).
5. Оцінку ефективності запропонованого рішення за критеріями затримки, пропускну здатності та використання ресурсів [6].

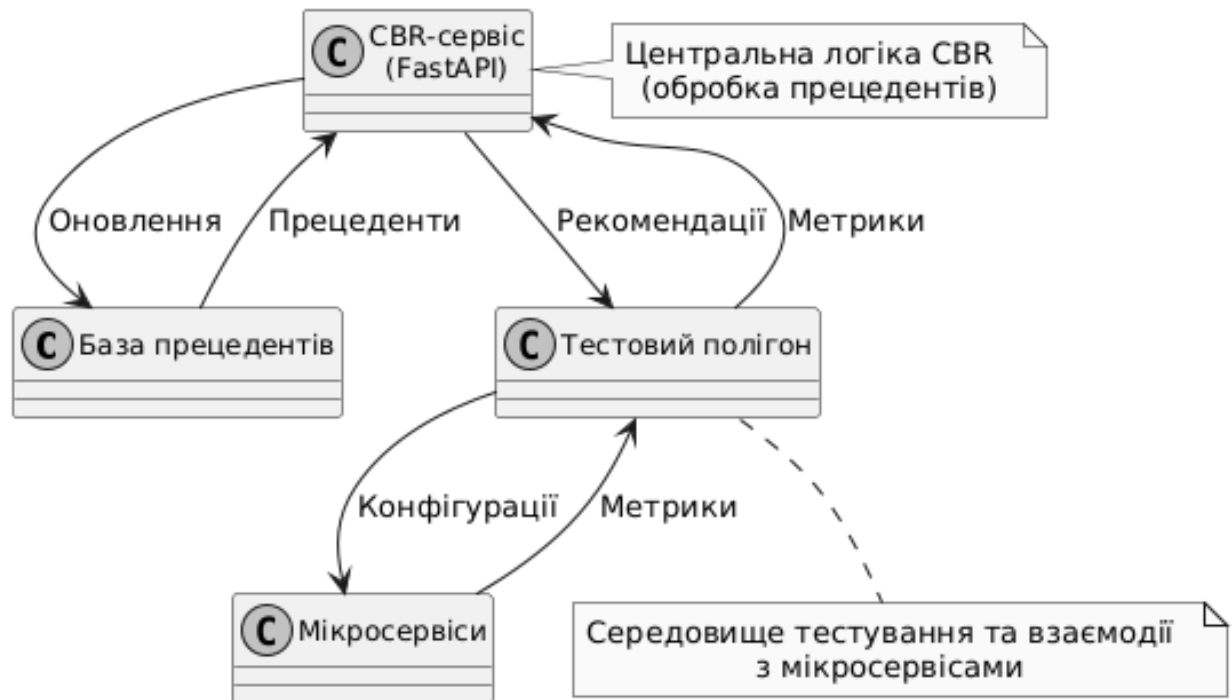


Рисунок 1.5 – Блок-схема архітектури запропонованого рішення

Очікується, що розроблений програмний засіб забезпечить адаптивне конфігурування мікросервісів у реальному часі, що підвищить продуктивність і стабільність систем у порівнянні з традиційними методами [5].

2 АНАЛІЗ ІНСТРУМЕНТІВ КОНФІГУРУВАННЯ ПРОГРАМНИХ МІКРОСЕРВІСІВ ТА ВИКОРИСТАННЯ МЕТОДІВ СВР ДЛЯ АУКР

2.1 Інструменти конфігурування програмних мікросервісів

В цьому параграфі наведений аналіз існуючих програмних рішень, які використовуються для конфігурування мікросервісів.

2.1.1 Аналіз Kubernetes ConfigMaps

Kubernetes ConfigMaps є компонентом платформи оркестрації контейнерів Kubernetes, що призначений для зберігання та управління конфігураційними даними мікросервісів. За допомогою ConfigMaps можемо відокремити конфігураційні параметри від коду додатків, що спрощує їх оновлення без необхідності модифікації образів контейнерів [9]. Основні функції цього компоненту включають в себе:

- зберігання даних у вигляді пар ключ-значення або цілих конфігураційних файлів у форматі YAML/JSON;
- монтування конфігурацій у контейнери як змінні середовища, аргументи командного рядка або файли в томах;
- оновлення конфігурацій через Kubernetes API або утиліту kubectl, що дозволяє змінювати параметри без перезапуску pod'ів у деяких випадках;
- інтеграція з Kubernetes Secrets для управління чутливими даними.

Оскільки ConfigMaps є компонентом Kubernetes, то не важко здогадатись, що його головною особливістю є тісна інтеграція з усією екосистемою Kubernetes, що дозволяє автоматично застосовувати оновлення конфігурацій до pod'ів через механізм rolling updates. Вони підтримують декларативний підхід, де конфігурації описуються в YAML-маніфестах, що полегшує автоматизацію розгортання. Крім того, ConfigMaps дозволяють одночасно управляти конфігураціями для кількох мікросервісів у кластері, забезпечуючи централізоване управління [9].

Очевидно, що в кожного з представлених в цьому параграфі інструментів є свої переваги та недоліки, і ConfigMaps не став виключенням. Інструмент має наступні переваги:

- висока масштабованість завдяки архітектурі Kubernetes, що підходить для великих кластерів із сотнями мікросервісів;
- простота інтеграції з Kubernetes-екосистемою, включаючи Helm і оператори;
- декларативний підхід спрощує автоматизацію та аудит змін;
- гнучкість у форматах даних (ключ-значення, файли).

Ну і, звичайно ж, не обійшлося і без недоліків, а саме:

- низька підтримка оновлення в реальному часі: зміни ConfigMaps не завжди миттєво застосовуються до всіх pod'ів через затримки в синхронізації;
- обмеження для середовищ поза Kubernetes, що робить інструмент непридатним для неконтейнерних систем;
- вимагає ручного оновлення конфігурацій або додаткових інструментів (наприклад, Kustomize) для автоматизації;
- складність управління при великій кількості ConfigMaps у складних проєктах.

Як інструмент, Kubernetes ConfigMaps широко використовуються в контейнерних додатках для управління конфігураціями мікросервісів, розгорнутих у Kubernetes-кластерах. Наприклад, вони застосовуються для налаштування підключення до баз даних (змінні типу DB_HOST, DB_PORT), управління API-ключами або конфігурації параметрів логування. ConfigMaps також використовуються для передачі конфігураційних файлів (наприклад, nginx.conf) до вебсерверів у pod'ах [9].

На рисунку 2.1 наведено схему монтування ConfigMap до pod'а в Kubernetes через механізм Kubernetes API, що ілюструє процес інтеграції конфігураційних даних у контейнер мікросервісу[9].

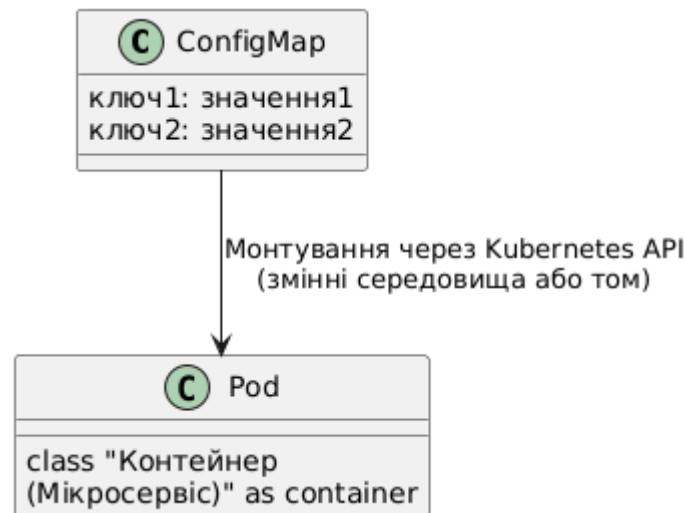


Рисунок 2.1 – Монтування ConfigMap до pod'а в Kubernetes

2.1.2 Аналіз HashiCorp Consul

HashiCorp Consul, в свою чергу, є розподіленою системою для управління конфігураціями, виявлення сервісів і моніторингу стану мікросервісів. Consul використовує сховище ключ-значення для зберігання конфігураційних даних, які доступні через HTTP API, CLI або вебінтерфейс [12]. Основні функціонал цієї системи в себе включає:

- централізоване зберігання конфігурацій із підтримкою динамічного оновлення;
- виявлення сервісів через DNS або API, що спрощує комунікацію між мікросервісами;
- моніторинг стану вузлів і сервісів за допомогою health checks;
- підтримка розподіленої синхронізації конфігурацій між дата-центрами.

Унікальність Consul проявляється у підтримці оновлення конфігурацій у реальному часі завдяки gossip-протоколу, який забезпечує миттєву синхронізацію між вузлами. Інструмент також пропонує вбудовану підтримку service mesh через Consul Connect, що дозволяє управляти безпечною комунікацією між мікросервісами [12].

Перевагами цього інструменту є:

- висока автоматизація завдяки API та динамічному оновленню конфігурацій;
- підтримка реального часу, що критично для високодинамічних систем;
- гнучкість для гетерогенних середовищ (не обмежується однією платформою);
- вбудована підтримка моніторингу та безпеки.

Недоліками ж виступають наступні проблеми та виклики: gossip-протокол може створювати накладні витрати в кластерах із тисячами вузлів; складність початкового налаштування, особливо для команд без досвіду роботи з Consul; залежність від інфраструктури Consul, що вимагає додаткових серверів; ну а також обмеження для дуже великих систем через продуктивність сховища ключ-значення.

Consul використовується в МСА, де потрібне динамічне управління конфігураціями та виявлення сервісів. Наприклад, інструмент застосовується для налаштування параметрів API-шлюзів, балансувальників навантаження або для синхронізації конфігурацій між регіонами в розподілених системах. Consul також популярний у service mesh сценаріях для управління трафіком між мікросервісами [12].

На рисунку 2.2 наведено схему синхронізації конфігурацій мікросервісів через HashiCorp Consul за допомогою gossip-протоколу, що ілюструє процес передачі даних від Consul Server до кількох мікросервісів [12].

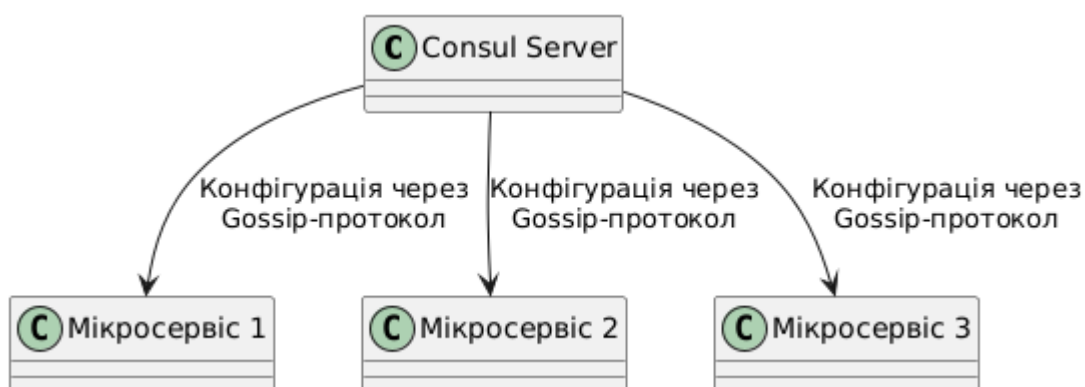


Рисунок 2.2 – Синхронізація конфігурацій через HashiCorp Consul

2.1.3 Аналіз Spring Cloud Config

Spring Cloud Config – це інструмент для централізованого управління конфігураціями в розподілених системах, розроблений для Java-додатків на основі Spring Boot. Він використовує Git-репозиторій для зберігання конфігураційних даних, які доставляються до мікросервісів через HTTP API [13]. Основні функції цього інструменту в себе включають:

- централізоване зберігання конфігурацій у Git із підтримкою версіонування;
- автоматичне оновлення клієнтських додатків при змінах у репозиторії через Spring Cloud Bus;
- підтримка профілів для різних середовищ (наприклад, development, production);
- інтеграція з Spring Security для захисту конфігурацій.

Як вже можна було побачити з переліку основного функціоналу Spring Cloud Config, його особливістю є тісна інтеграція з Git, що дозволяє версіонувати конфігурації, відстежувати зміни та забезпечувати аудит. Інструмент підтримує профілювання, що дає змогу адаптувати конфігурації до різних середовищ без модифікації коду [13].

З переваг можна виділити високу масштабованість, простоту інтеграції з Java-додатками на Spring Boot, прозорість і контроль змін завдяки використанню Git, ну а також гнучкість у налаштуванні профілів до різних середовищ.

З мінусів же, в свою чергу, я можу виділити наступне: середня підтримка реального часу (оновлення залежить від частоти Git pull або Spring Cloud Bus, що може створювати затримки); обмеження для всіх не-Java-систем, оскільки інструмент орієнтований саме на Spring-екосистему (як не важко здогадатись з його назви); потреба в додаткових компонентах (Spring Cloud Bus, RabbitMQ) для динамічного оновлення; складність управління при великій кількості конфігураційних файлів.

Знову ж таки, як не дивно, але Spring Cloud Config застосовується в Java-мікросервісах для управління конфігураціями, наприклад, налаштування підключення до баз даних, параметрів API або конфігурації кешування. Інструмент популярний у Spring Boot-додатках, де потрібне централізоване управління конфігураціями для десятків сервісів у розподіленій системі [13].

На рисунку 2.3 наведено схему архітектури Spring Cloud Config для конфігурування мікросервісів, яка ілюструє процес отримання конфігураційних даних із Git-репозиторію через Spring Cloud Config Server і передачу їх до мікросервісу за допомогою HTTP [13].

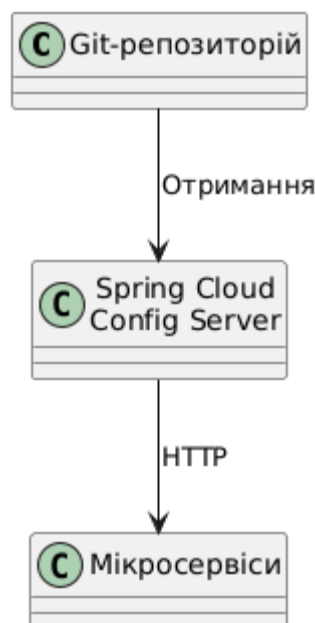


Рисунок 2.3 – Архітектура Spring Cloud Config для конфігурування мікросервісів

2.1.4 Аналіз AWS Config

AWS Config є хмарним сервісом від Amazon Web Services, який призначений для моніторингу, оцінки та управління конфігураціями хмарних ресурсів. Сервіс здатен автоматично відстежувати зміни в конфігураціях ресурсів (наприклад, EC2, S3) і забезпечувати їх відповідність заданим правилам [14]. Основні функції включають:

- постійний аудит конфігурацій із записом історії змін;

- оцінка відповідності стандартам безпеки та комплаєнсу (HIPAA, PCI DSS);
- інтеграція з AWS Lambda для автоматизації реакцій на зміни;
- надання звітів і аналітики через AWS Management Console або API.

AWS Config, на відміну від попередньо зазначених інструментів, вирізняється автоматизованим моніторингом із підтримкою правил відповідності, що дозволяє виявляти небажані зміни в реальному часі. Інструмент пропонує вбудовану інтеграцію з AWS CloudTrail для детального аудиту подій, що впливають на конфігурації [14].

Перевагами цього інструменту є його висока автоматизація, завдяки хмарній інфраструктурі AWS; висока масштабованість, що підходить для великих хмарних систем; простота інтеграції з іншими AWS-сервісами (як не дивно); а також підтримка безпеки та комплаєнсу через вбудовані правила.

Недоліками є середня підтримка реального часу через затримки в обробці подій (0.21 хвилина); обмеження для не-AWS середовищ; висока вартість при моніторингу великої кількості ресурсів; а також складність налаштування для новачків без досвіду роботи з AWS.

Зазвичай цей інструмент прийнято використовувати для управління конфігураціями хмарних мікросервісів, наприклад, відстеження змін у налаштуваннях EC2-інстансів, S3-бакетів або RDS-баз. Інструмент популярний у системах, що потребують відповідності стандартам безпеки, наприклад, для забезпечення шифрування даних або обмеження публічного доступу до ресурсів [14].

На рисунку 2.4 наведено схему моніторингу хмарних ресурсів (EC2, RDS, S3) за допомогою AWS Config, яка ілюструє процес збору даних про конфігурації через безперервний моніторинг, їх аналіз на відповідність заданим правилам і передачу результатів до журнальної аудиторії для подальшого аналізу [14].

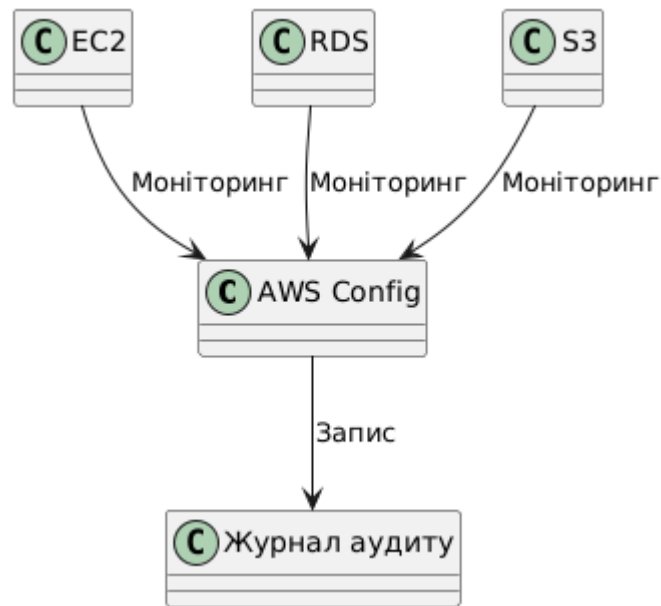


Рисунок 2.4 – Моніторинг ресурсів за допомогою AWS Config

2.2 Використання методів CBR для АУКМ

І вже тепер ми можемо перейти до більш детального ознайомлення з методами CBR, які знадобляться нам для виконання цього дослідження.

2.2.1 Загальний принцип роботи CBR у контексті АУКР

Метод CBR ґрунтується на вирішенні нових задач шляхом використання досвіду з попередніх ситуацій, що робить його ефективним для АУКР. У цьому контексті, CBR дозволяє автоматично підбирати оптимальні конфігурації для мікросервісів на основі поточного стану системи, такого як завантаження процесора, використання пам'яті чи затримка відповіді [2].

Основою CBR є цикл R4, який складається з чотирьох етапів: вилучення (Retrieve) найближчого прецеденту з БД, повторне використання (Reuse) його рішення для поточної задачі, перегляд (Revise) рішення для адаптації до нових умов і збереження (Retain) оновленого прецеденту для майбутнього використання. Для АУКР прецеденти містять метрики системи (наприклад, `cpu_load`, `response_time`) та відповідні конфігурації сервісів, такі як розмір пулу підключень до БД або ліміти запитів. Адаптація CBR для конфігурування мікросервісів передбачає зіставлення поточного стану системи з історичними

прецедентами, вибір конфігурації, що мінімізує затримки або витрати ресурсів, і її застосування через API сервісів. Цей підхід забезпечує динамічне налаштування системи в реальному часі, що особливо важливо для розподілених мікросервісних архітектур.

На рисунку 2.5 наведена схема основи CBR – циклу R4.

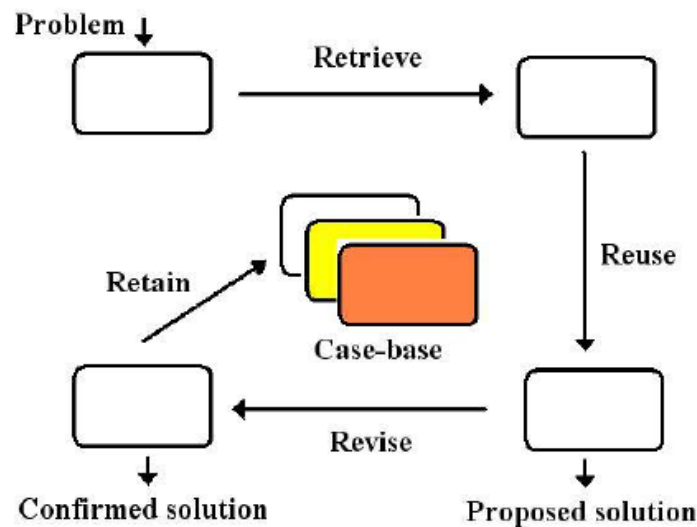


Рисунок 2.5 – Цикл R4

2.2.2 Аналіз K-Nearest Neighbors

Алгоритм k-найближчих сусідів є одним із найпростіших методів вилучення прецедентів у CBR, який базується на пошуку k найближчих прецедентів до поточного стану системи за метрикою відстані [6]. У контексті АУКР KNN порівнює вектор поточних метрик системи (наприклад, `cpu_load`, `memory_usage`, `performance_metric`) із векторами історичних прецедентів, обираючи конфігурацію, що відповідає найближчому або найпоширенішому рішенню серед k сусідів. Структура алгоритму включає нормалізацію даних для уніфікації масштабів метрик, обчислення відстаней між векторами та вибір k найближчих прецедентів. Математична модель KNN використовує евклідову відстань для оцінки близькості між поточним станом $S = (s_1, s_2, s_3, \dots, s_n)$ і прецедентом $C = (c_1, c_2, c_3, \dots, c_n)$:

$$d(S, C) = \sqrt{\sum_{i=1}^n (s_i - c_i)^2}, \quad (2.1)$$

де S – вектор поточного стану системи,

C – вектор прецеденту,

$d(S, C)$ – евклідова відстань між векторами S і C ,

n – кількість метрик або розмірність векторів S і C ,

s_i – значення i -ї метрики поточного стану,

c_i – значення i -ї метрики прецеденту.

Після визначення k сусідів алгоритм обирає конфігурацію шляхом голосування або усереднення параметрів. Перевагою KNN є його простота реалізації та здатність адаптуватися до змін у системі без складного навчання, однак алгоритм має високу обчислювальну складність $O(n)$ для великих баз прецедентів, чутливість до шуму в даних і не враховує різну важливість ознак. У АУКР KNN ефективно застосовується для вибору конфігурацій у стабільних умовах.

2.2.3 Аналіз Weighted K-Nearest Neighbors

Weighted KNN є модифікацією KNN, яка присвоює вагу кожному прецеденту залежно від його близькості до поточного стану системи, що підвищує точність вибору конфігурації [6]. На відміну від стандартного KNN, де всі k сусідів мають однаковий вплив, Weighted KNN надає більшу вагу ближчим прецедентам, що дозволяє краще враховувати релевантність даних. Структура алгоритму подібна до KNN, але включає додатковий етап обчислення ваг. Математична модель передбачає використання ваги w для прецеденту i , яка обернено пропорційна відстані:

$$w_i = \frac{1}{d(S,C) + \epsilon}, \epsilon > 0, \quad (2.2)$$

де w_i – вага, присвоєна i -му прецеденту,

ϵ – запобігає діленню на нуль.

Конфігурація обирається шляхом зваженого голосування:

$$Config = \arg \max_{conf} \sum_{i \in k\text{-nearest}} w_i * \mathbb{I}(conf_i = conf), \quad (2.3)$$

де $Config$ – оптимальна конфігурація,

$\arg \max_{\text{conf}}$ – оператор, що визначає конфіг, який максимізує суму ваг,
 $\|$ – індикаторна функція,
 conf_i – конфігурація, що асоційована з i -м прецедентом,
 conf – кандидатська конфігурація, можливе значення конфігурації.

Перевагою Weighted KNN є вища точність завдяки врахуванню релевантності прецедентів і гнучкість у налаштуванні ваг, однак алгоритм залишається обчислювально складним і потребує ретельного вибору параметра ϵ . У АУКР Weighted KNN застосовується, коли окремі метрики, такі як `performance_metric`, мають більшу вагу, наприклад, для налаштування `rate_limit_requests` у системах із високим навантаженням, де затримка є критичною.

2.2.4 Аналіз Feature-Based Retrieval

Feature-Based Retrieval вилучає прецеденти на основі відповідності ключових ознак поточного стану системи, а не загальної відстані між векторами, що робить його ефективним для специфічних задач АУКР [2]. Алгоритм фокусується на окремих характеристиках, таких як тип сервісу, рівень навантаження або конкретні метрики (наприклад, `availability`), і оцінює їхню подібність. Структура алгоритму включає визначення набору ознак, обчислення функції подібності для кожної ознаки та вибір прецеденту з найвищою сумарною подібністю. Математична модель базується на зваженій сумі подібностей:

$$\text{Similarity}(S, C) = \sum_{i=1}^n w_i * \text{sim}(s_i, c_i), \quad (2.4)$$

де $\text{sim}(s_i, c_i)$ – функція подібності для ознаки i .

Перевагою методу є висока точність для задач із чітко визначеними ознаками та можливість ігнорувати нерелевантні метрики, однак він вимагає ретельного підбору ознак і ваг, а також може бути менш ефективним для складних систем із великою кількістю ознак. У АУКР Feature-Based Retrieval використовується для вибору конфігурацій за специфічними критеріями,

наприклад, налаштування `db_connection_type` для сервісів із високим `response_time`, коли інші метрики менш важливі.

2.2.5 Аналіз Cluster-Based Retrieval

Cluster-Based Retrieval групує прецеденти в кластери за схожими характеристиками, використовуючи алгоритми кластеризації, такі як K-Means, і вибирає прецеденти з найближчого кластера, що значно прискорює пошук у великих базах [6]. У контексті АУКР алгоритм спочатку кластеризує прецеденти за метриками (`cpu_load`, `memory_usage` тощо), потім визначає кластер, найближчий до поточного стану системи, і обирає конфігурацію з цього кластера. Структура алгоритму включає етап кластеризації, обчислення відстані до центроїдів і вибір прецедентів. Математична модель K-Means мінімізує суму квадратів відстаней до центроїдів:

$$J = \sum_{i=1}^k \sum_{x \in C_i} \|x - \mu_i\|^2, \quad (2.5)$$

де J – функція втрат,

x – вектор характеристик окремого прецеденту в кластері C_i ,

C_i – множина всіх прецедентів, що входять до i -го кластера,

μ_i – центроїд кластера C_i .

Перевагою методу є швидкість пошуку в великих базах і стійкість до шуму, однак він залежить від правильного вибору кількості кластерів k і може втрачати точність при неоднорідних даних. У АУКР Cluster-Based Retrieval застосовується для швидкого вибору конфігурацій у системах із великою кількістю прецедентів, наприклад, налаштування `max_products_per_page` для кластера високого навантаження.

2.2.6 Аналіз Indexing and Hashing

Indexing and Hashing використовує хеш-функції для створення індексів, що прискорюють пошук прецедентів у великих базах, роблячи метод особливо ефективним для АУКР у реальному часі [6]. Алгоритм створює індекси на основі ключових метрик, таких як `cpu_load` або `performance_metric`, групуючи

прецеденти в «бакети» за значеннями хеш-функції. Структура алгоритму включає етап індексації, де прецеденти розподіляються за хеш-ключами, і етап пошуку, де поточний стан системи хешується для швидкого доступу до відповідного бакета. Математична модель базується на хеш-функції h , яка відображає прецедент C у ключ:

$$h(C) = \text{hash}(c_1, c_2, \dots, c_n), \quad (2.6)$$

де c_n – компонент вектора характеристик прецеденту,

hash – хеш-функція, що перетворює вектор характеристик прецеденту в унікальний числовий або символічний ключ.

Пошук виконується за індексом, що забезпечує середню складність $O(1)$. Перевагою методу є висока швидкість пошуку та ефективність для великих баз, однак він може втрачати точність через колізії хеш-функції та потребує ретельного налаштування хешування для складних ознак. У AUKP Indexing and Hashing застосовується для швидкого вибору конфігурацій у реальному часі, наприклад, налаштування `rate_limit_requests` за хешем `performance_metric` у високодинамічних системах.

Для реалізації цих методів в коді було використано бібліотеки `sklearn`, `numpy` та `math` у Python. Для Cluster-Based Retrieval було застосовано алгоритм K-Means із бібліотеки `sklearn.cluster` для кластеризації прецедентів, а також бібліотеку `numpy` для обчислення евклідових відстаней і роботи з масивами даних (див. лістинг A.1 в додатку A). Для Feature-Based Retrieval використано бібліотеку `math` для обчислення евклідової відстані між випадками на основі вибраних ознак (див. лістинг A.2 в додатку A). У Indexing and Hashing застосовано бібліотеку `numpy` для генерації випадкових площин і обчислень у реалізації Locality-Sensitive Hashing (LSH), а також кастомний клас `LSHIndex` для індексації та пошуку (див. лістинг A.3 в додатку A). В методах найближчих сусідів було використано `math` для обчислення евклідової відстані між випадками (див. лістинг A.4 в додатку A).

3 ПРОЄКТУВАННЯ ТЕСТОВОГО ПОЛІГОНУ ТА СИСТЕМИ АУКМ

3.1 Вимоги до тестового полігону та системи АУКМ

В 3 главі були розглянуті питання проектування тестового полігону та системи АУКМ. Були сформульовані функціональні та нефункціональні вимоги для тестового полігону та системи АУКМ.

3.1.1 Функціональні вимоги

Функціональні вимоги визначають основні можливості тестового полігону та АУКМ, що забезпечують їхню роботу в умовах моделювання реальних сценаріїв.

Тестовий полігон призначений для симуляції мікросервісної архітектури, що складається з трьох основних мікросервісів: `auth_service` для автентифікації користувачів, `product_service` для управління каталогом продуктів і `order_service` для обробки замовлень. Полігон має підтримувати генерацію різного рівня (низького, середнього, високого), а також забезпечувати можливість ручного налаштування конфігураційних параметрів, наприклад, розміру пулу підключень до бази даних або лімітів запитів. Для цього полігон повинен надавати API-інтерфейси для взаємодії з мікросервісами, що дозволяють отримувати метрики та застосовувати конфігурації [3].

Система АУКМ, у свою чергу, відповідає за збирання метрик, таких як завантаження процесора, використання пам'яті, затримка відповіді та доступність, автоматичний підбір оптимальних конфігурацій на основі цих самих метрик системи, використовуючи методи CBR, такі як K-Nearest Neighbors або Cluster-Based Retrieval тощо. АУКМ має забезпечити обробку прецедентів, що включає вилучення релевантного прецеденту з бази даних, адаптацію конфігурації до поточного стану та збереження нових прецедентів для подальшого навчання.

Крім того, система повинна інтегруватися з тестовим полігоном через API, дозволяючи застосовувати конфігурації до мікросервісів у реальному часі, а також надавати адміністратору інтерфейс для налаштування параметрів алгоритмів CBR, наприклад, кількості сусідів у KNN [2].

Тож, у висновку – тестовий полігон має підтримувати симуляцію навантаження, а також можливість конфігурування кожного окремого мікросервісу, в той час як система АУКМ має підтримувати збирання та відстеження метрик, забезпечувати автоматичний підбір конфігурацій через CBR, ну а також інтеграцію з мікросервісами та збереження прецедентів.

3.1.1 Нефункціональні вимоги

Нефункціональні вимоги визначають якісні характеристики тестового та системи АУКМ, які забезпечують їхню ефективність, надійність і придатність для практичного використання.

Щодо продуктивності, тестовий полігон має обробляти до 1000 одночасних запитів із середньою затримкою не більше 200 мс, а АУКМ повинна виконувати підбір конфігурації за 0.5 секунди для забезпечення адаптивності в реальному часі [20].

Масштабування системи є критичним аспектом: тестовий полігон має підтримувати до 10 мікросервісів із можливістю додавання нових без модифікації архітектури, тоді як АУКМ повинна масштабуватися для обробки бази до 2000 прецедентів без значного зростання затримки.

Надійність забезпечується стійкістю до збоїв: тестовий полігон має зберігати стабільність при 95% доступності, а АУКМ – гарантувати коректну роботу при тимчасовій недоступності одного з мікросервісів. Безпека включає захист конфігураційних даних через авторизацію за протоколом OAuth 2.0 для API-доступу та шифрування даних у базах із використанням AES-256 [15].

Простота інтеграції досягається завдяки використанню RESTful API для зв'язків між компонентами, що дозволяє легко підключати нові сервіси або алгоритми CBR. Розширюваність системи забезпечується модульною

архітектурою, яка дозволяє додавати нові метрики, наприклад, `operational_costs`, або методи CBR, такі як `Indexing and Hashing`, без значних змін у структурі [6].

В таблиці 3.1 наведені сформульовані функціональні і нефункціональні вимоги для до тестового полігону та системи АУКМ.

Таблиця 3.1. Функціональні та нефункціональні вимоги до тестового полігону та системи АУКМ

Вимога	Тестовий полігон	Система АУКМ
Функціональні	Симуляція навантаження, збирання метрик, API для конфігурування	Автоматичний підбір конфігурацій через CBR, інтеграція з API, збереження прецедентів
Продуктивність	Затримка до 200 мс, обробка 1000 запитів	Підбір конфігурації за 0.5 с
Масштабованість	Підтримка до 10 мікросервісів	Обробка до 2000 прецедентів
Надійність	95% доступність	Стійкість до збоїв одного мікросервісу
Безпека	OAuth 2.0, шифрування AES-256	OAuth 2.0, шифрування AES-256
Інтеграція/Розширюваність	RESTful API, модульна архітектура	RESTful API, підтримка нових методів CBR

3.2 Проєктування тестового полігону

І вже тепер ми можемо перейти до проєктування наших систем. Першим на черзі виступить тестовий полігон, який загалом буде складатись з трьох мікросервісів, про які я розкажу детальніше нижче.

3.2.1 Опис мікросервісів тестового полігону

Тестовий полігон моделює мікросервісну архітектуру, що складається з трьох основних сервісів: `auth_service`, `product_service` та `order_service`, які імітують типові функції веб-додатка для електронної торгівлі. Кожен сервіс

виконує чітко визначені завдання, забезпечуючи модульність і можливість адаптивного управління конфігураціями [3].

Auth_service відповідає за автентифікацію та управління користувачами, включаючи реєстрацію, вхід і видачу токенів доступу на основі протоколу OAuth 2.0.

Product_service забезпечує управління каталогом продуктів, дозволяючи переглядати, додавати та видаляти продукти, а також обробляти запити на пошук.

Order_service відповідає за створення, оновлення та відстеження замовлень, включаючи обробку даних про транзакції та статуси.

Кожен мікросервіс має власну базу даних PostgreSQL для ізоляції даних, а взаємодія між ними здійснюється через RESTful API. Для адаптивного управління кожен сервіс підтримує набір конфігураційних параметрів, які АУКМ може змінювати, наприклад, `token_expiry` і `max_login_attempts` для *auth_service*, `rate_limit_requests` і `db_pool_size` для *product_service*, `max_orders_per_page` і `default_order_timeout` для *order_service*. Ці параметри дозволяють адаптувати поведінку сервісів до різних рівнів навантаження, від низького до пікового, що забезпечує гнучкість тестування [20]. На рисунку 3.1 наведена схема архітектури полігону.

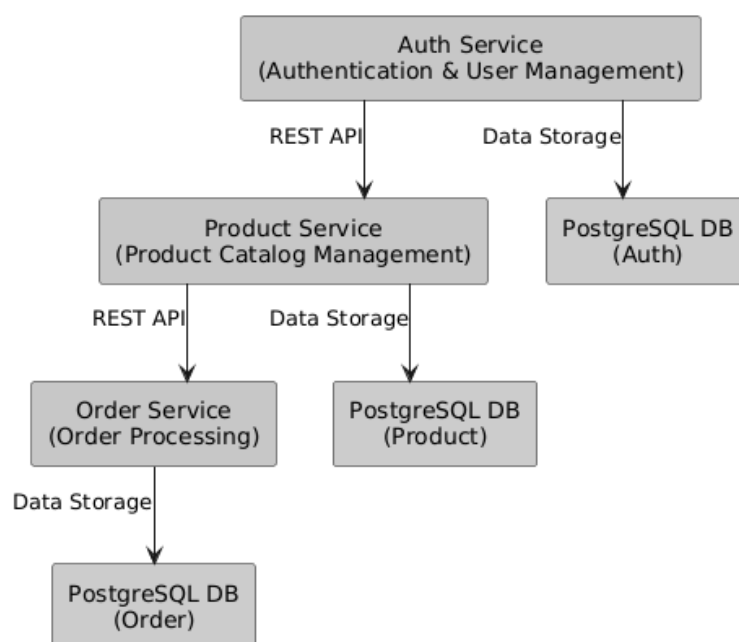


Рисунок 3.1 – Схема архітектури тестового полігону

3.2.2 Діаграма прецедентів

На рисунку 3.2 наведена діаграма прецедентів, яка відображає взаємодію користувачів із тестовим полігоном, враховуючи три ролі: неавторизований користувач, зареєстрований користувач та адміністратор.

Неавторизований користувач має доступ до перегляду каталогу продуктів через `product_service`, але не може виконувати дії, що потребують автентифікації, наприклад, створювати замовлення.

Зареєстрований користувач, після автентифікації через `auth_service`, може переглядати продукти, створювати та відстежувати замовлення через `order_service`, а також оновлювати свій профіль.

Адміністратор, окрім можливостей зареєстрованого користувача, має доступ до управління каталогом продуктів (додавання, видалення) і конфігураціями мікросервісів, наприклад, зміна лімітів запитів або розміру пулу підключень.

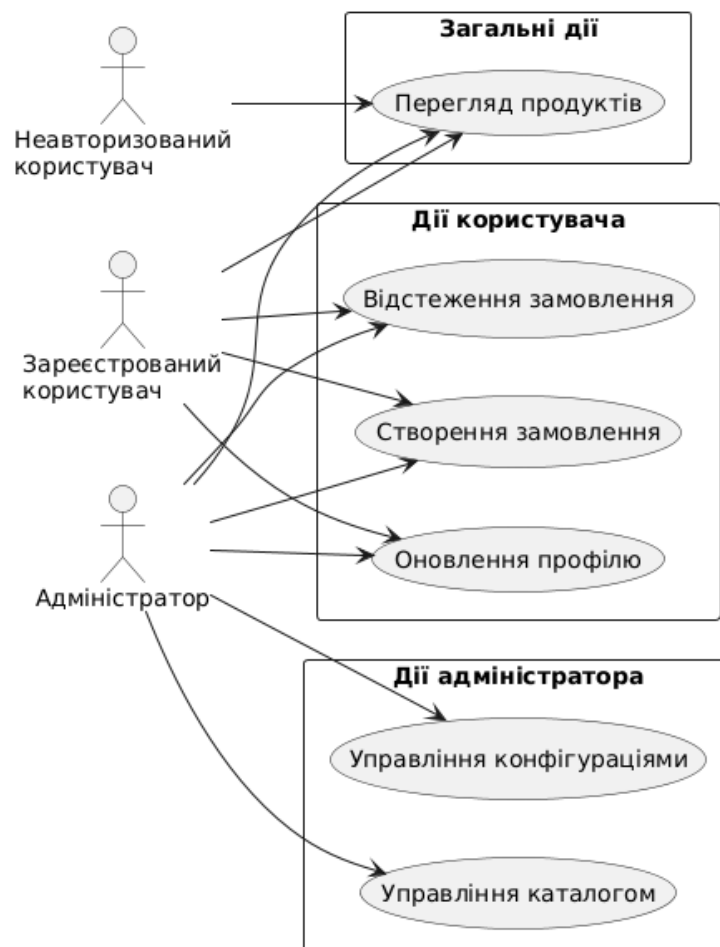


Рисунок 3.2 – Діаграма прецедентів тестового полігону

Усі взаємодії здійснюються через API-ендпоінти, що забезпечує чітке розмежування прав доступу [21]. Діаграма прецедентів допомагає визначити функціональні вимоги до системи та спрощує проєктування інтерфейсів.

3.2.3 Структура бази даних

Структура бази даних тестового полігону (див. рис. 3.3) розроблена для забезпечення ізоляції даних кожного мікросервісу, що відповідає принципам мікросервісної архітектури. Кожен сервіс (auth_service, product_service, order_service) використовує окрему схему в PostgreSQL із набором таблиць, які відображають їхню функціональність [22].

Загалом хочу зазначити, що взаємодія між цими базами буде відбуватись через відповідні ендпоінти у маршрутизації кожного з мікросервісів, проте наразі, для більшої наочності, зв'язки будуть продемонстровані таким чином, наче це одна БД.

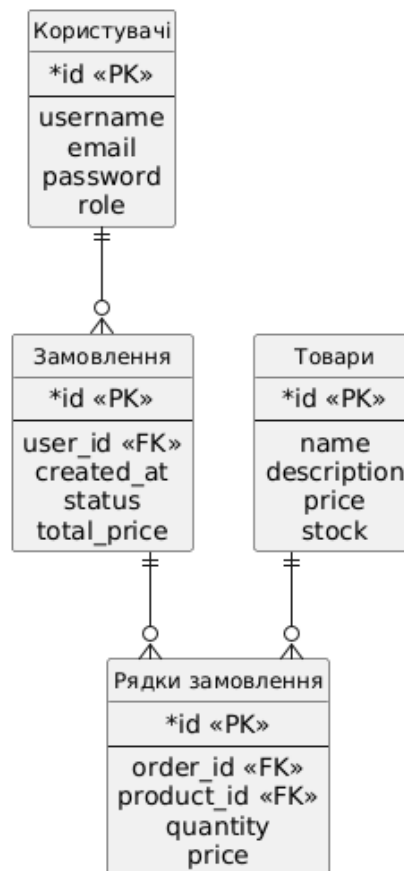


Рисунок 3.3 – ER-діаграма бази даних тестового полігону

Для сервісу авторизації передбачено таблицю `users` з полями `ID` (первинний ключ), `username`, `email`, `password`, а також `role` (буде визначатись як `ENUM` звиклого користувача, або адміністратора). Токен сесії буде передаватись через `cookie`, тому не бачу сенсу додавати його в базу.

Для сервісу продуктів передбачено таблицю `products` із полями `ID` (первинний ключ), `name`, `description`, `price` та `stock`. В цілому, тут також немає нічого важкого та складного, бо знову ж таки, зазначу – це всього лиш тестовий полігон, який не потребує важких та замудрьонних структур.

Ну і на останок, сервіс для замовлень складається з двох таблиць, а саме: `orders`, що складається з `ID` (первинний ключ), `user_id` (зовнішній ключ), `created_at`, `status` та `total price`; а також `order_items`, що представляє собою рядок замовлення, що складається з `ID` (первинний ключ), `order_id` (зовнішній ключ) б `product_id` (зовнішній ключ), `quantity` та `price`.

3.2.4 Архітектура та зв'язки мікросервісів

Архітектура тестового полігону базується на розподіленій МСА, де кожен сервіс – `auth_service`, `product_service` та `order_service` – є автономним і будуть розгортатись в контейнерах `Docker` (див. рис. 3.4).

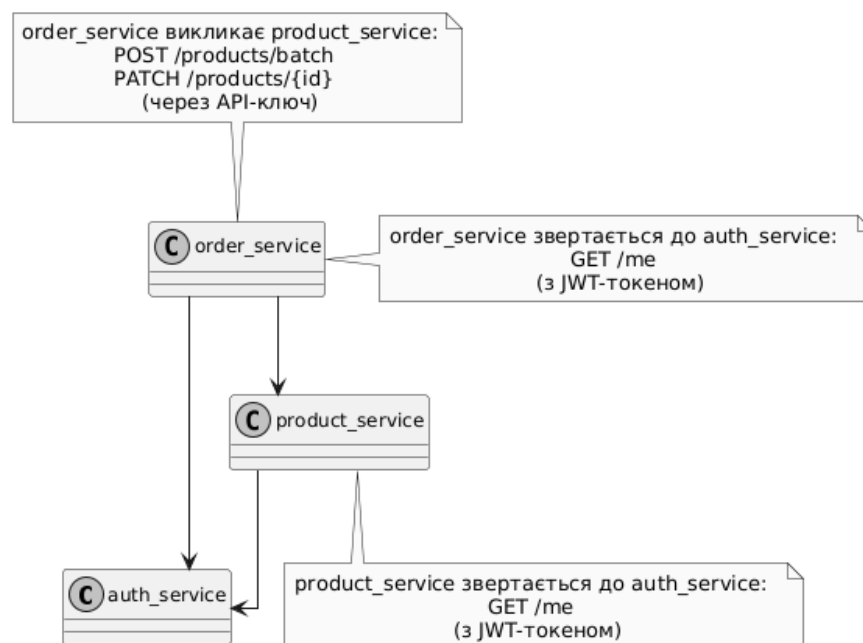


Рисунок 3.4 – Архітурна діаграма тестового полігону із зв'язками між мікросервісами

На відміну від систем із централізованим API-шлюзом, взаємодія між мікросервісами здійснюватиметься напряду через синхронні RESTful API-запити, що спрощує архітектуру, але вимагає ретельного управління автентифікацією та обмеженнями доступу [3].

Auth_service надає ендпоінти для автентифікації, такі як POST /login для видачі JWT-токенів і POST /register для створення користувачів, а також захищені ендпоінти для адміністраторів, наприклад, GET /users і POST /config/update.

Product_service підтримує операції з каталогом через ендпоінти GET /products, GET /products/{product_id} для перегляду продуктів і POST /products для їх створення (лише для адміністраторів), а також PATCH /products/{product_id} для оновлення запасів, доступний для order_service через API-ключ.

Order_service обробляє замовлення через POST /orders для створення, GET /my-orders для перегляду користувацьких замовлень і GET /orders для адміністраторів, взаємодіючи з product_service через POST /products/batch для перевірки наявності продуктів [20].

Автентифікація між сервісами забезпечується передачею JWT-токенів у заголовках Authorization, які product_service і order_service перевіряють, звертаючись до auth_service через GET /me.

3.2.5 Плановані конфігураційні параметри

Тестовий полігон підтримує набір конфігураційних параметрів для кожного мікросервісу, які система АУКМ адаптивно змінюватиме на основі метрик системи, таких як завантаження процесора чи затримка відповіді.

Для auth_service передбачено наступні параметри: access_token_expire_minutes (тривалість дії токена, 15-60 хвилин), algorithm (алгоритм шифрування JWT, HS256 або RS256), db_connection_type (тип підключення до бази, sync або async), db_pool_size (розмір пулу підключень,

3-15), `rate_limit_requests` (ліміт запитів за хвилину, 50-200), `response_delay` (штучна затримка відповіді, 0.0-1.0 секунди).

`Product_service` включає `db_connection_type` (`sync` або `async`), `db_pool_size` (3-15), `rate_limit_requests` (50-200), `response_delay` (0.0-1.0 секунди), `max_products_per_page` (кількість продуктів на сторінці, 10-50).

`Order_service` має `db_connection_type` (`sync` або `async`), `db_pool_size` (3-15), `rate_limit_requests` (50-200), `response_delay` (0.0-1.0 секунди), `max_orders_per_page` (кількість замовлень на сторінці, 10-50), `default_order_status` (статус замовлення за замовчуванням, `pending` або `processing`) [20].

Ці параметри дозволяють адаптувати сервіси до різних сценаріїв навантаження: низьке (до 100 запитів/с), середнє (100-500 запитів/с), високе (500-1000 запитів/с) і пікове (понад 1000 запитів/с). Зміна параметрів планується через захищені ендпоінти `POST /config/update`, доступні лише для адміністраторів, із подальшим перезапуском бази даних у разі зміни `db_connection_type` або `db_pool_size`.

3.3 Проектування системи АУКМ

Тепер можемо перейти до головної частини цієї роботи – проектуванню та подальшій реалізації системи АУКМ.

3.3.1 Опис компонентів системи АУКМ

Система АУКМ призначена для автоматичного підбору оптимальних конфігурацій тестового полігону на основі метрик системи, використовуючи методи CBR (див. рис. 3.5). Система складається з трьох основних компонентів: модуль моніторингу, CBR-модуль і модуль конфігурування, які взаємодіють для забезпечення адаптивності [2].

Модуль моніторингу відповідає за збір метрик із тестового полігону, таких як завантаження процесора (`cpu_load`), використання пам'яті (`memory_usage`), дискового простору (`disk_usage`), мережевий трафік

(network_usage), середня затримка (performance_metric), час відповіді (response_time), операційні витрати (operational_costs) і доступність (availability). Цей модуль планується інтегрувати з мікросервісами через ендпоінти /health, використовуючи HTTP-запити для отримання даних про стан.

CBR-модуль обробляє зібрані метрики, порівнюючи поточний стан із історичними прецедентами, збереженими в базі даних, і вибирає оптимальну конфігурацію за допомогою методів KNN, Weighted KNN, Feature-Based Retrieval, Cluster-Based Retrieval або Indexing and Hashing.

Модуль конфігурування застосовує обрані конфігурації до мікросервісів (auth_service, product_service, order_service) через ендпоінти POST /config/update, використовуючи JWT-токени для авторизації. Крім того, АУКМ передбачає збереження нових прецедентів у базі даних, включаючи метрики, конфігурації та показники ефективності, для подальшого навчання [6].

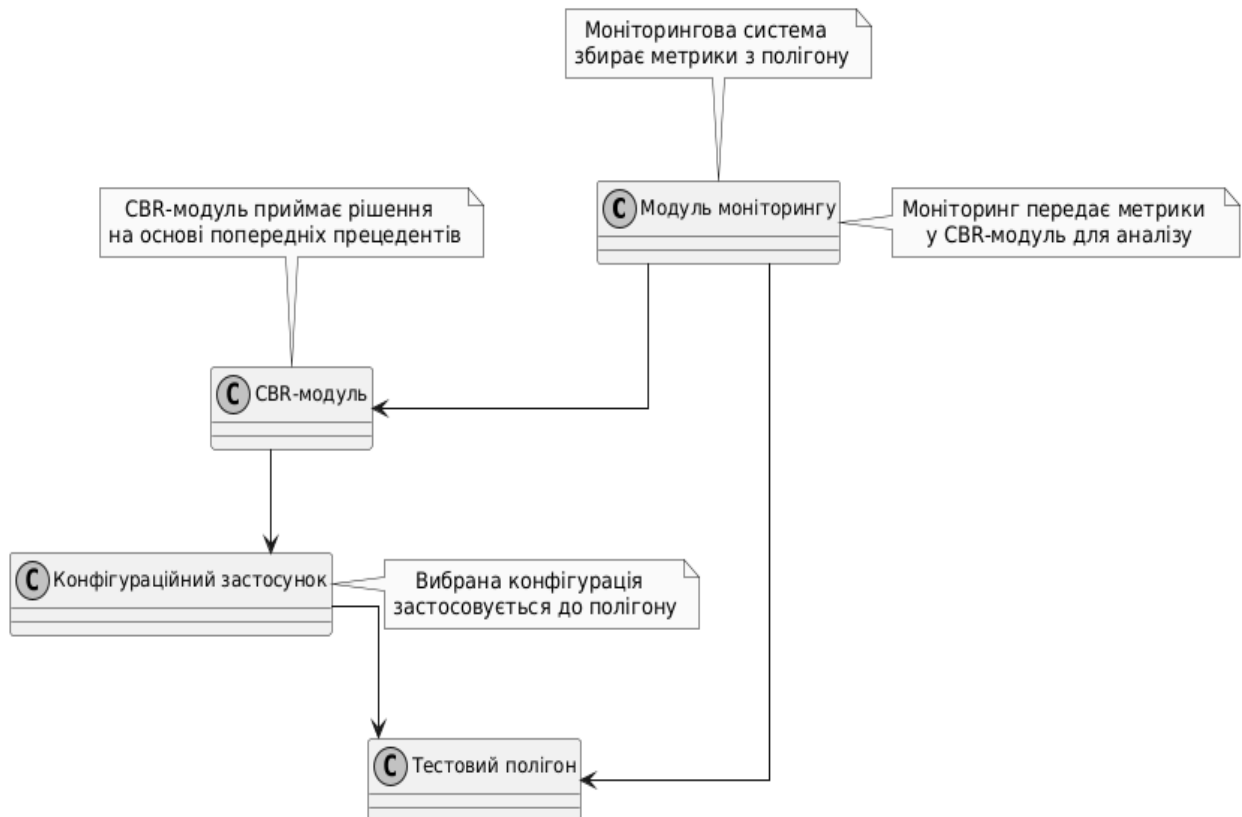


Рисунок 3.5 – Діаграма взаємодії компонентів системи АУКМ

3.3.2 Діаграма прецедентів для системи АУКМ

На рисунку 3.6 наведена діаграма прецедентів, яка відображає взаємодію користувачів і автоматичних процесів із системою АУКМ. Основними акторами є адміністратор і автоматична система.

Адміністратор має доступ до налаштування параметрів CBR-алгоритмів через ендпоінт GET /auto-config, де він може вибрати метод (knn, weighted_knn, feature_based, cluster_based, indexing_hashing), задати ваги для Weighted KNN, ознаки для Feature-Based Retrieval або кількість кластерів для Cluster-Based Retrieval. Адміністратор також може переглядати поточні конфігурації всіх сервісів через GET /configs і оновлювати їх вручну через PUT /update-config/{service}.

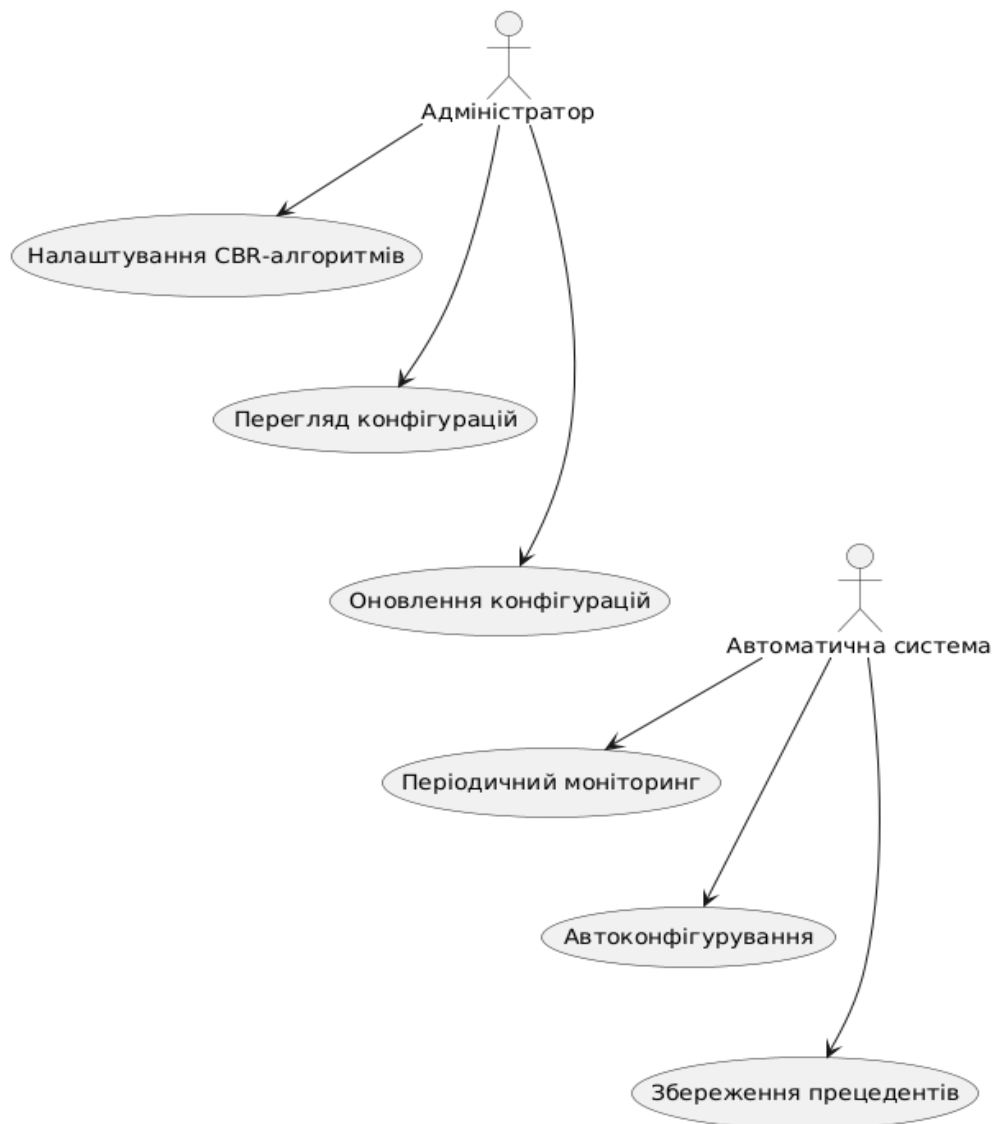


Рисунок 3.6 – Діаграма прецедентів для системи АУКМ

Автоматична система виконує періодичний моніторинг стану тестового полігону через WebSocket-ендпоінт /ws, запускає автоконфігурування за розкладом, зберігає нові прецеденти в базу даних і застосовує конфігурації. Ці дії забезпечують безперервну адаптацію системи до змін навантаження без втручання користувача [21]. Діаграма прецедентів допомагає чітко визначити ролі та функціональні можливості АУКМ.

3.3.3 Структура бази даних системи АУКМ

На рисунку 3.7 наведена база даних системи АУКМ, яка розроблена для зберігання історичних прецедентів, які використовуються CBR-модулем для вибору оптимальних конфігурацій. Основною таблицею є cases, яка містить поля: ID (первинний ключ), timestamp (час створення прецеденту), cpu_load, memory_usage, disk_usage, network_usage, performance_metric, response_time, operational_costs, availability (метрики системи), configs (JSON-рядок із конфігураціями сервісів), adaptation_time (час адаптації) і reconfigurations (кількість реконфігурацій).

Прецеденти (cases)
*id «ПК»
timestamp
cpu_load
memory_usage
disk_usage
network_usage
performance_metric
response_time
operational_costs
availability
configs
adaptation_time
reconfigurations

Рисунок 3.7 – ER-діаграма бази даних системи АУКМ

Таблиця не має зовнішніх ключів, оскільки прецеденти є ізольованими записами, що спрощує масштабування [22]. Для оптимізації запитів планується створити індекси на поля timestamp і performance_metric, які часто

використовуються для пошуку та сортування. База даних використовуватиме PostgreSQL, ізольовану від баз тестового полігону, для забезпечення автономності АУКМ.

3.3.4 Архітектура та зв'язки системи АУКМ

Система АУКМ проєктується як централізований мікросервіс, який інтегрується з тестовим полігоном через RESTful API та WebSocket для моніторингу в реальному часі [3]. Як я вже встиг зазначити раніше, архітектура включає три компоненти: модуль моніторингу, CBR-модуль і модуль конфігурування, які взаємодіють із базою даних PostgreSQL через ORM (див. рис. 3.8).

Модуль моніторингу періодично надсилає HTTP-запити до ендпоінтів /health мікросервісів (auth_service, product_service, order_service) для збору метрик, а також використовує WebSocket-ендпоінт /ws для безперервного оновлення даних. CBR-модуль обробляє метрики, застосовуючи один із методів (KNN, Weighted KNN, Feature-Based, Cluster-Based, Indexing and Hashing) для вибору конфігурації з таблиці cases. Модуль конфігурування надсилає конфігурації до мікросервісів через POST /config/update, використовуючи JWT-токени, отримані через POST /login від auth_service. АУКМ розгортатиметься в контейнерах Docker.

3.3.5 Інтеграція АУКМ із тестовим полігоном

Інтеграція системи АУКМ із мікросервісним полігоном здійснюється через REST API та WebSocket для моніторингу та конфігурування в реальному часі. На рисунку 3.9 подано діаграму взаємодії, де модуль моніторингу надсилає запити до /health для збору метрик, а WebSocket-канал /ws забезпечує їх регулярне оновлення. Для доступу до захищених ендпоінтів АУКМ використовує JWT-токен, отриманий від auth_service [20].

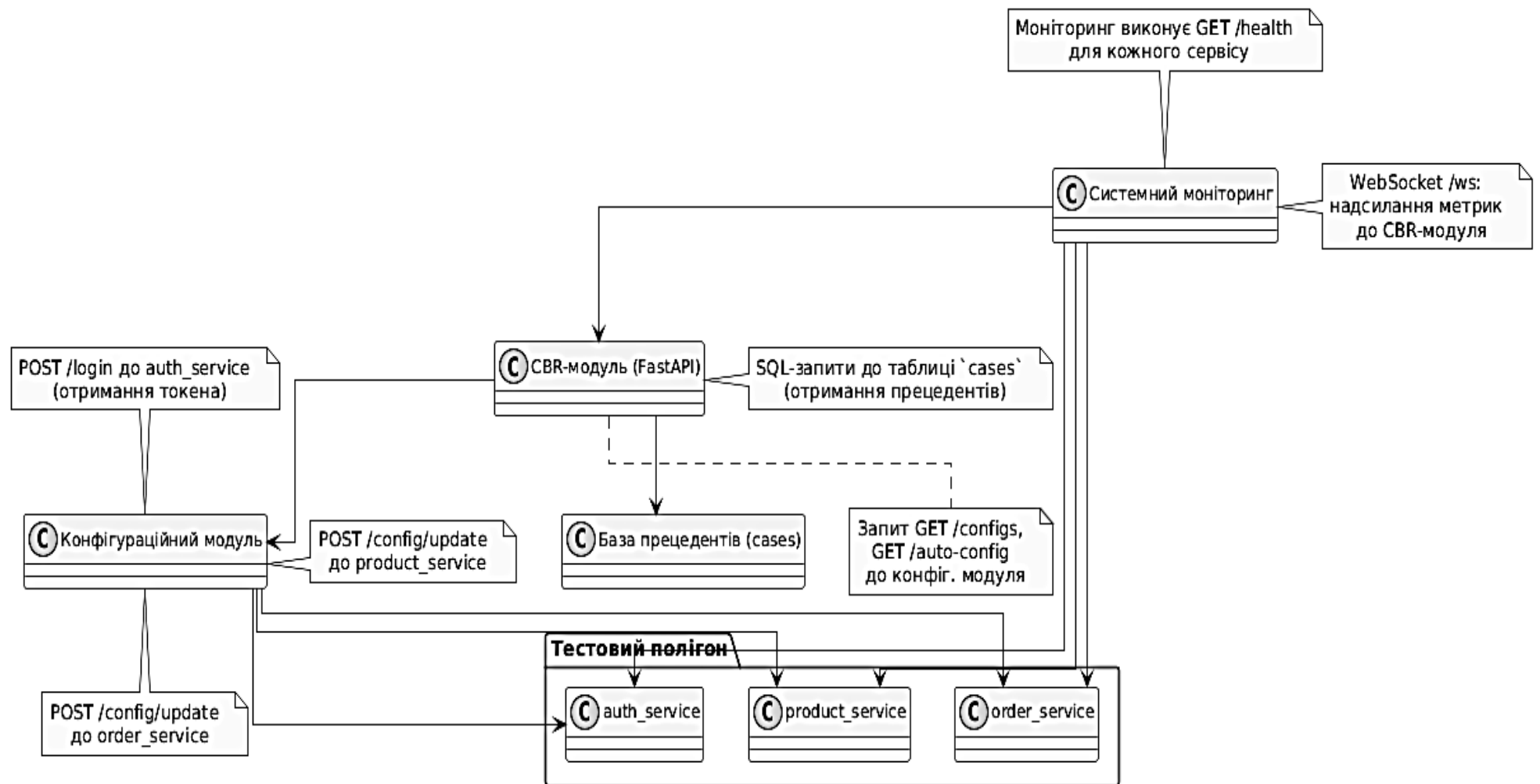


Рисунок 3.8 – Архітектурна діаграма системи АУКМ із зв'язками до тестового полігону

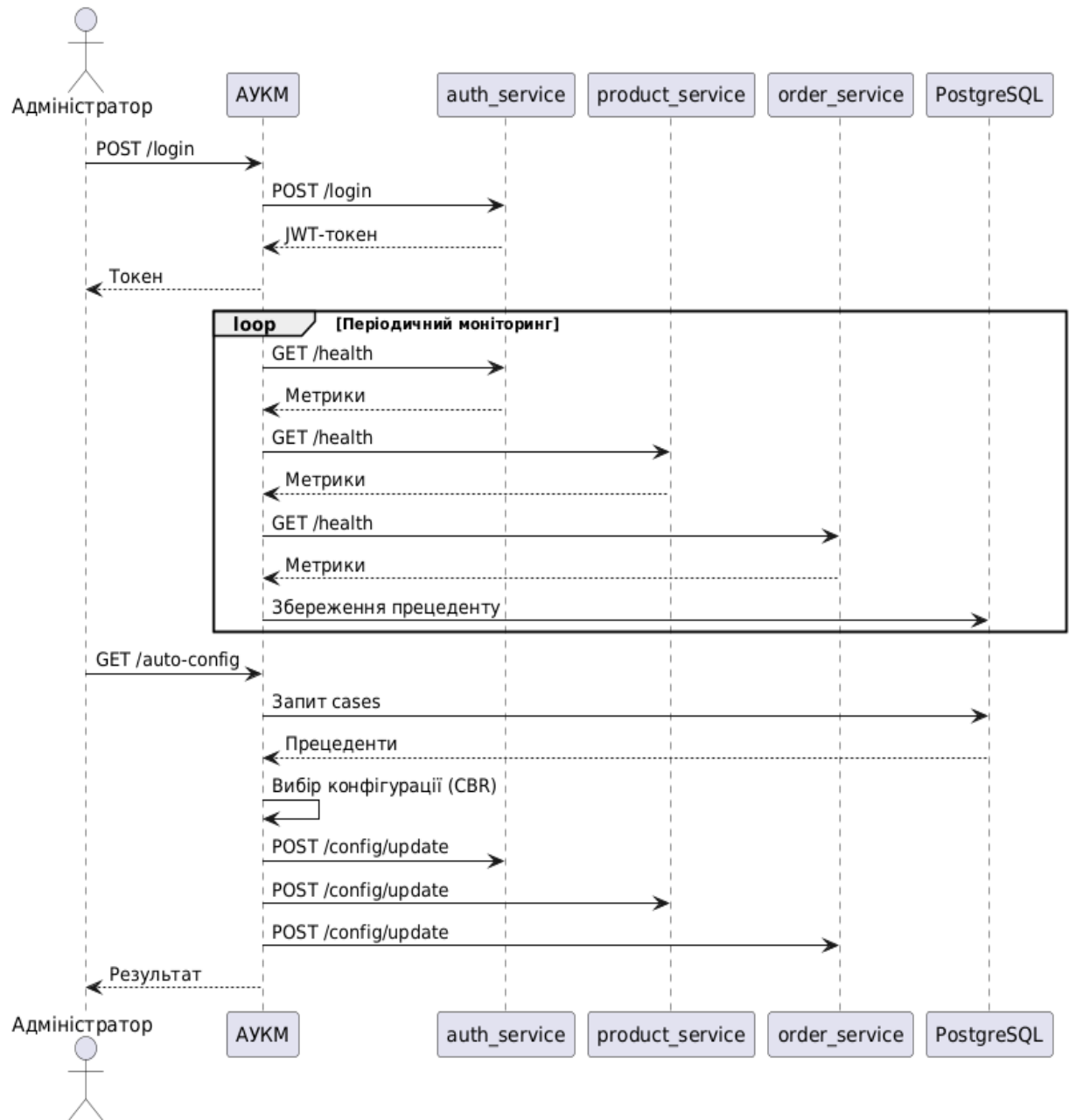


Рисунок 3.9 – Діаграма взаємодії АУКМ із мікросервісами полігону

CBR-модуль реалізує CBR-методи для вибору конфігурацій із таблиці cases за поточними метриками. Модуль конфігурування надсилає обрані параметри через API-запити. Нові прецеденти з результатами адаптації зберігаються в базі даних АУКМ для подальшого аналізу.

4 ПРОГРАМНА РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ ІНСТРУМЕНТУ АУКМ НА ТЕСТОВОМУ ПОЛІГОНІ

4.1 Технологічний стек реалізації

Для реалізації мікросервісного тестового застосунку (тобто тестового полігону) було обрано мову програмування Python, а саме фреймворк FastAPI. Цей фреймворк було обрано через підтримку асинхронного програмування, високу швидкість обробки запитів і зручність створення RESTful API [16]. FastAPI забезпечує автоматичну генерацію документації API через OpenAPI, що полегшує тестування та інтеграцію (саме цим інструментом, SwaggerUI, я користувався для тестування мікросервісного застосунку).

В якості БД для абсолютно кожного мікросервісу було обрано PostgreSQL в ізольованих схемах, що гарантує незалежність і безпеку даних [22]. PostgreSQL обрано за його надійність, підтримку складних запитів і широке поширення в мікросервісних архітектурах.

Для ізоляції кожного окремого мікросервісу було використано Docker. Саме це забезпечує однакове середовище на етапах розробки, тестування та розгортання. Docker Compose використовується для локального розгортання, дозволяючи швидко налаштувати мережу та залежності між сервісами [17].

В цілому, дуже схожа ситуація і з самим АУКМ – мова та фреймворк лишились незмінними, БД також, інструмент для контейнеризації той самий. З цікавого – було використано бібліотеки scikit-learn (для реалізації методів KNN та K-Means для кластеризації), а також NumPy для обробки векторів метрик [23].

Інструменти розробки включають PyCharm як основне середовище розробки, pytest для створення юніт-тестів, а також Docker Desktop для зручнішого керування контейнерами та імейджами ізольованих середовищ мікросервісів.

4.2 Реалізація тестового полігону

Спершу нам варто ознайомитись з тим, як саме я реалізував тестовий полігон, на якому і будуть проходити усі випробування. Як можна було зрозуміти з попередніх розділів, полігон складається з трьох зв'язаних між собою мікросервісів: мікросервіс для авторизації, роботи з продуктами, а також для роботи з замовленнями. Зараз ми познайомимось з кожним з них ближче.

4.2.1 Реалізація `auth_service`

Мікросервіс `auth_service` є ключовим компонентом тестового полігону, відповідаючи за автентифікацію користувачів, управління їхніми профілями, видачу JWT-токенів і конфігурування сервісу.

Основна функціональність `auth_service` зосереджена на обробці запитів для реєстрації, авторизації, адміністрування користувачів і динамічного налаштування параметрів роботи сервісу, таких як тривалість дії токенів чи розмір пулу підключень до бази даних. Також не варто забувати про генерацію JWT-токенів для авторизації в інших мікросервісах (`product_service`, `order_service`).

Модель бази даних реалізовано через SQLAlchemy, що забезпечує гнучке управління даними [18]. Паролі користувачів хешуються за допомогою бібліотеки `passlib` із алгоритмом `bcrypt`, а JWT-токени генеруються бібліотекою `python-jose` з використанням секретного ключа та конфігураційного алгоритму (за замовчуванням HS256) [24].

Основні маршрути `auth_service`, реалізовані в модулі `routes.py`, забезпечують взаємодію з клієнтами та іншими мікросервісами:

- POST `/login` – авторизує користувача за електронною поштою та паролем, повертаючи JWT-токен, який зберігається в `cookie` для подальшої авторизації;
- POST `/register` – створює нового користувача з хешованим паролем і роллю за замовчуванням «user»;

- POST /config/update – оновлює конфігураційні параметри (наприклад, access_token_expire_minutes, db_pool_size), доступний лише адміністраторам;
- GET /config – повертає поточну конфігурацію сервісу (для адміністраторів);
- GET /me – повертає дані поточного користувача на основі JWT-токена;
- GET /users – повертає список усіх користувачів (для адміністраторів);
- POST /promote/{user_id} – підвищує користувача до ролі адміністратора;
- POST /logout – видаляє, завершує сесію.

Для обмеження кількості запитів використовується бібліотека `slowapi`, яка застосовує динамічний ліміт (`rate_limit_requests`) із конфігурації [25]. Конфігураційні параметри, визначені в класі `AuthConfig` (`config.py`), дозволяють налаштувати тривалість дії токена (15-60 хвилин), алгоритм шифрування (HS256 або RS256), тип підключення до бази даних (синхронне або асинхронне), розмір пулу підключень (3-15) і затримку відповіді (0.0-1.0 секунди). Зміна параметрів, таких як `db_connection_type`, викликає переініціалізацію бази даних через функцію `reinitialize_db` (`database.py`).

Контейнеризація `auth_service` реалізована через `Docker`. `Dockerfile` базується на образі `python:3.11-slim`, встановлює залежності з `requirements.txt` (`FastAPI`, `SQLAlchemy`, `passlib`, `python-jose` тощо) і запускає сервіс через `Uvicorn` на порті 8000. Використання `slim` образу дозволяє зменшити розмір контейнера, що прискорює його розгортання та знижує навантаження на інфраструктуру. Логування запитів і помилок забезпечується стандартною бібліотекою `logging`, що дозволяє відстежувати стан сервісу. Для моніторингу передбачено ендпоінт `GET /health`, який повертає статус «healthy» і використовується системою АУКМ для збору метрик. Цей ендпоінт є критично важливим для систем оркестрації та моніторингу, оскільки він надає простий механізм для перевірки працездатності сервісу, що є основою для подальшого збору детальних метрик системою АУКМ.

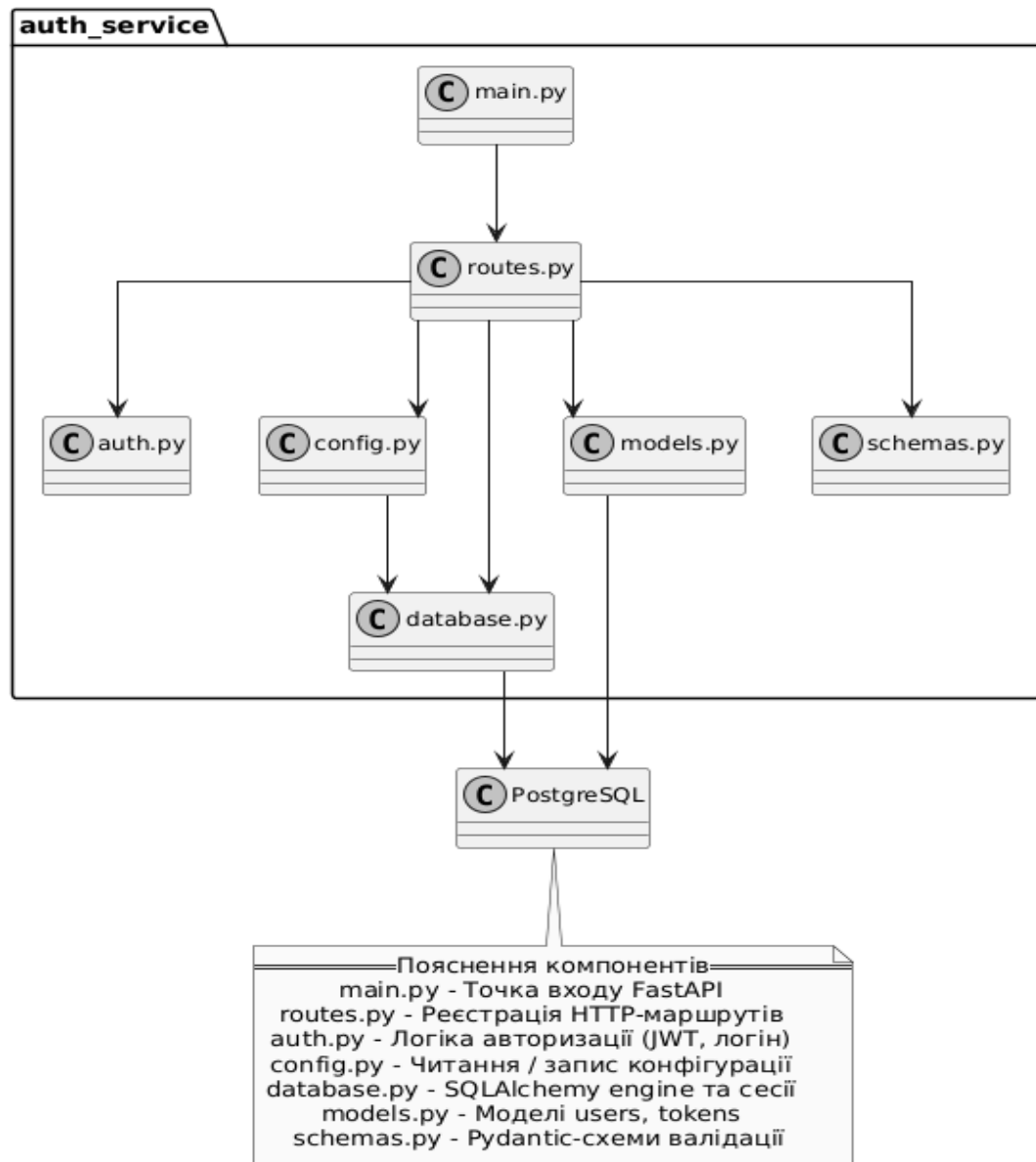


Рисунок 4.1 – Структура auth_service

4.2.2 Реалізація product_service

Мікросервіс product_service відповідає за управління каталогом продуктів, включаючи їх створення, перегляд, оновлення запасів і підтримку конфігураційних параметрів. Основна мета сервісу – надавати функціонал для роботи з продуктами, забезпечуючи інтеграцію з order_service для обробки замовлень і з auth_service для авторизації адміністративних дій. Функціонал product_service охоплює перегляд списку продуктів із підтримкою пошуку, пагінації та сортування, створення нових продуктів адміністраторами, а також оновлення запасів продуктів, що використовується order_service через

захищений API-ключ. Для зберігання даних застосовується PostgreSQL із таблицею `products`, яка містить поля: ідентифікатор, назву, опис, ціну, кількість на складі та шлях до зображення. Для асинхронних HTTP-запитів до `auth_service` (наприклад, перевірки ролі адміністратора) використовується бібліотека `aiohhttp` [26].

Основні маршрути `product_service`, як не дивно, знову визначені в модулі `routes.py`, забезпечують взаємодію з клієнтами та іншими сервісами:

- GET `/products` – повертає список продуктів із підтримкою пошуку за назвою, пагінації (`max_products_per_page`) та сортування за назвою, ціною чи запасами;
- GET `/products/{product_id}` – повертає деталі конкретного продукту за ідентифікатором;
- POST `/products` – створює новий продукт (доступно адміністраторам після авторизації через `auth_service`);
- PATCH `/products/{product_id}` – оновлює запаси продукту, доступне для `order_service` через API-ключ;
- POST `/products/batch` – повертає групу продуктів за списком ідентифікаторів, використовується `order_service`;
- POST `/config/update` – оновлює конфігурацію сервісу (наприклад, `db_pool_size`, `response_delay`), доступне адміністраторам;
- GET `/config` – повертає поточну конфігурацію (для адміністраторів).

Для обмеження запитів застосовується бібліотека `slowapi` з динамічним лімітом (`rate_limit_requests`), визначеним у конфігурації [25]. Конфігураційні параметри, реалізовані в класі `ProductConfig` (`config.py`), дозволяють налаштувати тип підключення до бази даних (синхронне або асинхронне), розмір пулу підключень (3-15), затримку відповіді (0.0-1.0 секунди), ліміт запитів (50-200 за хвилину) і максимальну кількість продуктів на сторінці (10-50). Зміна параметрів `db_connection_type` або `db_pool_size` викликає переініціалізацію бази даних через функцію `reinitialize_db` (`database.py`).

Контейнеризація `product_service` реалізована через Docker. Dockerfile, аналогічний до `auth_service`, базується на образі `python:3.11-slim`, встановлює залежності з `requirements.txt` (FastAPI, SQLAlchemy, aiohttp, slowapi тощо) і запускає сервіс через Uvicorn на порті 8002. Зміна порту порівняно з `auth_service` (8000) забезпечує ізоляцію сервісів у контейнерній мережі. Для моніторингу стану сервісу передбачено ендпоінт `GET /health`, який повертає статус «healthy» і використовується системою АУКМ для збору метрик, таких як завантаження процесора чи час відповіді [17].

Для кращого розуміння структури `product_service`, схему його компонентів можна побачити нижче на рисунку 4.2. Вона відображає організацію модулів і їх взаємодію з базою даних. Зазначу, що структура така сама як і в `auth_service`, тому на цій схемі відсутні додаткові підписи (деталі дивись на рисунку 4.1).

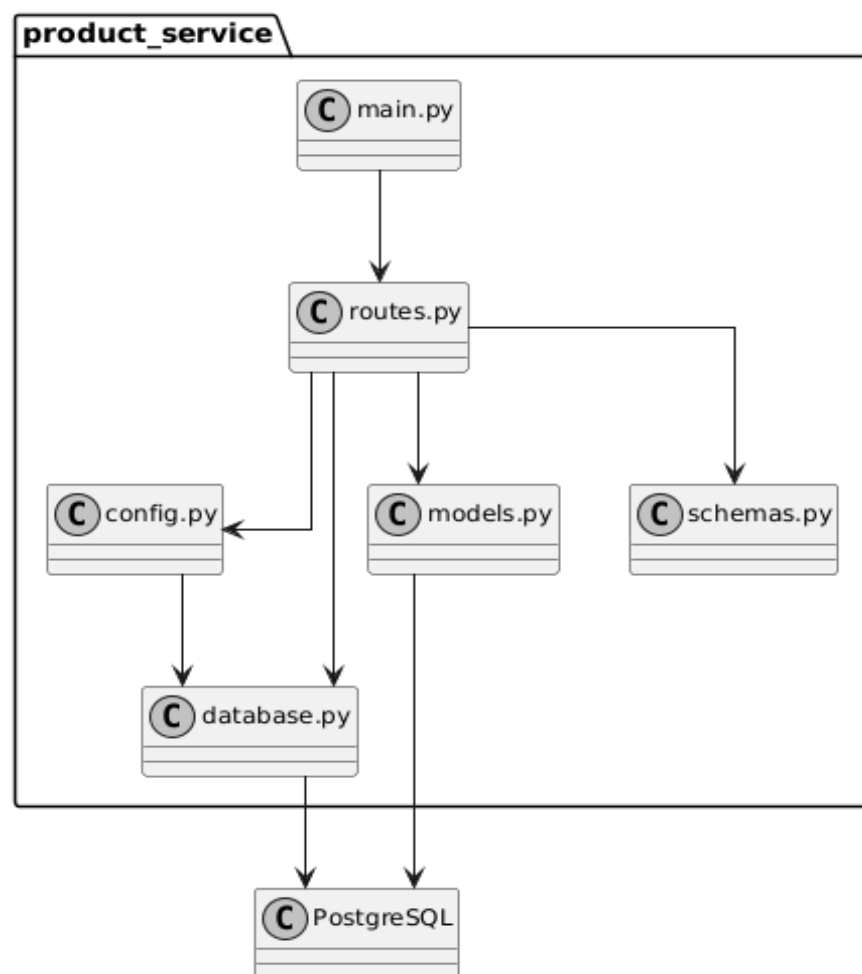


Рисунок 4.2 – Структура `product_service`

4.2.3 Реалізація order_service

Мікросервіс `order_service` є завершальним компонентом тестового полігону, який відповідає за обробку замовлень користувачів, їх створення, перегляд і управління конфігураційними параметрами. Завдяки використанню FastAPI в нас забезпечується швидка обробка запитів і асинхронна взаємодія з іншими мікросервісами, зокрема `auth_service` для авторизації та `product_service` для перевірки запасів продуктів [16]. Основна мета `order_service` – забезпечити надійне оформлення замовлень із автоматичним оновленням складських запасів і підтримкою гнучкого налаштування сервісу.

Основна функціональність `order_service` полягає в створенні замовлень, які включають список товарів, перевірку їхньої наявності через `product_service` і оновлення запасів після успішного оформлення. Користувачі можуть переглядати власні замовлення, тоді як адміністратори мають доступ до всіх замовлень у системі. Для зберігання даних використовується PostgreSQL із двома таблицями: `orders` (містить ідентифікатор замовлення, дату створення, статус, загальну ціну та ідентифікатор користувача) та `order_items` (зберігає деталі позицій замовлення: ідентифікатор продукту, кількість, ціну та прив'язку до замовлення). Моделі даних реалізовано через SQLAlchemy, що спрощує роботу з базою даних [18].

Для інтеграції з іншими мікросервісами `order_service` використовує асинхронні HTTP-запити через бібліотеку `aiohhttp`. Наприклад, при створенні замовлення сервіс звертається до `product_service` (POST `/products/batch`) для отримання даних про продукти та їхні запаси, а також до `auth_service` (GET `/me`) для перевірки авторизації користувача [26]. Ціни продуктів фіксуються на момент замовлення, що дозволяє уникнути невідповідностей при зміні цін у `product_service`.

Маршрути, реалізовані в модулі `routes.py`, забезпечують взаємодію з клієнтами та іншими сервісами:

- POST /orders – створює нове замовлення для авторизованого користувача, перевіряє наявність продуктів і оновлює запаси через product_service;
- GET /my-orders – повертає список замовлень поточного користувача з пагінацією (max_orders_per_page);
- GET /orders – повертає всі замовлення з пагінацією (доступно адміністраторам);
- GET /orders/{order_id} – повертає деталі конкретного замовлення, доступні власнику або адміністратору;
- POST /config/update – оновлює конфігурацію сервісу (наприклад, default_order_status, db_pool_size), доступне адміністраторам;
- GET /config – повертає поточну конфігурацію (для адміністраторів).

Для захисту від надмірних запитів використовується бібліотека slowapi з динамічним лімітом (rate_limit_requests), визначеним у конфігурації [25].

Конфігураційні параметри сервісу визначено в класі OrderConfig (config.py), який дозволяє налаштувати тип підключення до бази даних (синхронне або асинхронне), розмір пулу підключень (3-15), затримку відповіді (0.0-1.0 секунди), ліміт запитів (50-200 за хвилину), максимальну кількість замовлень на сторінці (10-50) та статус замовлення за замовчуванням (pending, processing тощо). Зміна параметрів db_connection_type або db_pool_size викликає переініціалізацію бази даних через функцію reinitialize_db (database.py). Логування запитів і помилок забезпечує відстеження стану сервісу, що є важливим для подальшого моніторингу системою АУКМ.

Контейнеризація order_service аналогічна до auth_service та product_service, з використанням Docker на базі образу python:3.11-slim. Сервіс запускається через Uvicorn на порті 8001, із залежностями, визначеними в requirements.txt (FastAPI, SQLAlchemy, aiohttp, slowapi тощо). Для моніторингу стану передбачено ендпоінт GET /health, який повертає статус «healthy» і використовується АУКМ для збору метрик [17].

4.2.4 Взаємодія між сервісами

Мікросервіси тестового полігону (`auth_service`, `product_service`, `order_service`) взаємодіють між собою через прямі HTTP-запити, забезпечуючи виконання основних сценаріїв, таких як авторизація користувачів, управління продуктами та створення замовлень. Комунікація побудована на асинхронних RESTful API-запитах, що використовують бібліотеку `aiohhttp` для ефективної обробки [26]. Кожен запит супроводжується механізмами авторизації, які базуються на JWT-токенах для користувацьких дій і API-ключах для міжсервісної взаємодії.

Основні потоки взаємодії включають:

- `order_service` → `product_service`: При створенні замовлення (`POST /orders`) `order_service` надсилає запит `POST /products/batch` до `product_service`, передаючи список ідентифікаторів продуктів. Це дозволяє отримати дані про ціни та запаси, необхідні для перевірки можливості оформлення замовлення. Після підтвердження `order_service` оновлює запаси через `PATCH /products/{product_id}`, використовуючи API-ключ для авторизації;
- `product_service` → `auth_service`: Для адміністративних дій, таких як створення продукту (`POST /products`) або оновлення конфігурації (`POST /config/update`), `product_service` надсилає запит `GET /me` до `auth_service`, передаючи JWT-токен у заголовок `Authorization`. Це перевіряє, чи має користувач роль адміністратора;
- `order_service` → `auth_service`: Аналогічно, `order_service` використовує `GET /me` для перевірки авторизації користувача під час створення замовлення (`POST /orders`) або перегляду власних замовлень (`GET /my-orders`).

Авторизація між сервісами забезпечується двома механізмами. JWT-токени, згенеровані `auth_service` (`POST /login`), використовуються для ідентифікації користувачів і перевірки їхніх ролей. Токени передаються в `cookie` або заголовок `Authorization` і перевіряються через `auth_service`. API-

ключі застосовуються для захисту міжсервісних запитів, наприклад, при оновленні запасів продуктів (PATCH /products/{product_id}), де order_service передає ключ у заголовок X-API-Key. Ці механізми забезпечують безпеку та ізоляцію сервісів [1].

Для наочності взаємодія між мікросервісами зображена на рисунку 4.3, де показано основні HTTP-запити та механізми авторизації.

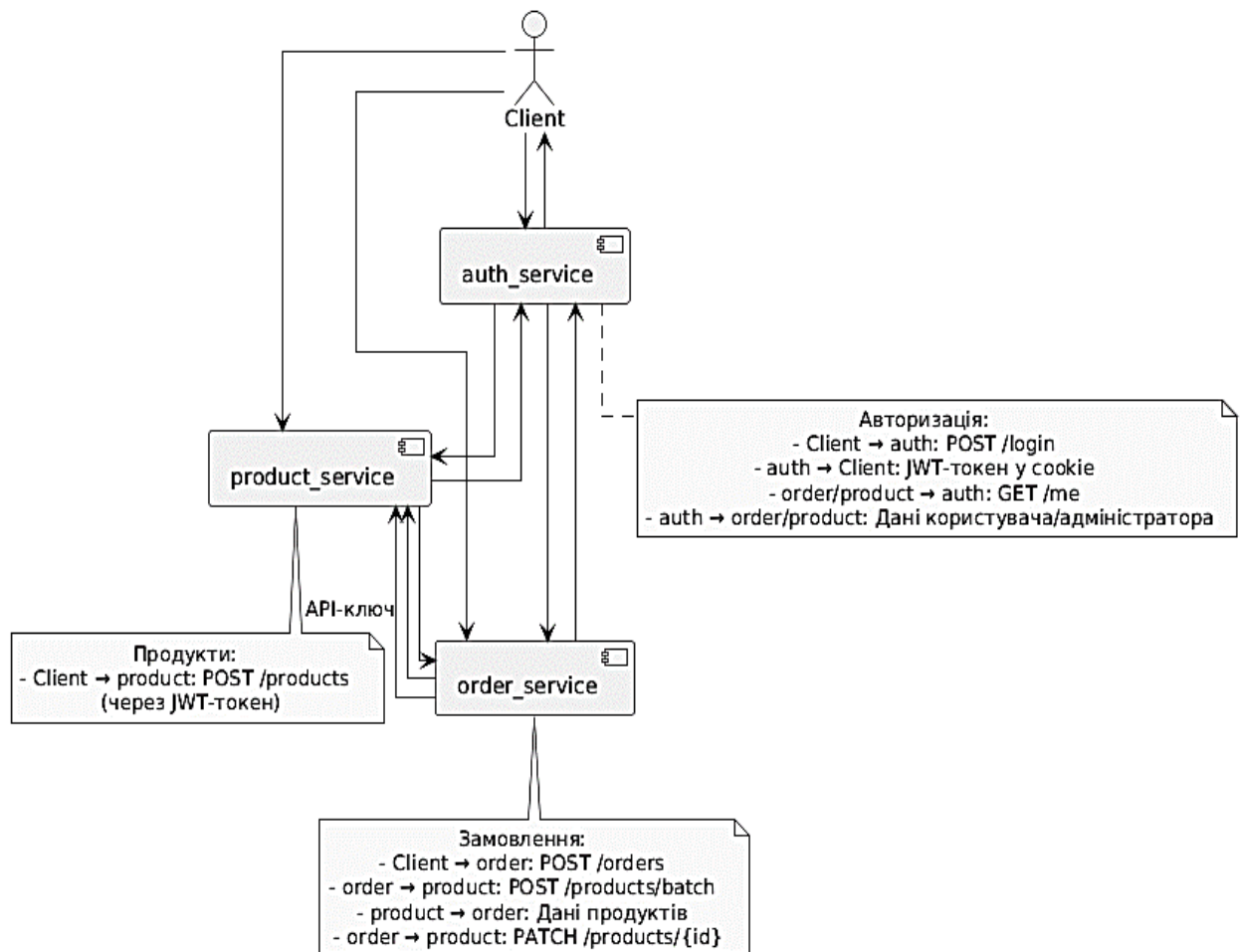


Рисунок 4.3 – Взаємодія між мікросервісами тестового полігону

4.3 Контейнеризація тестового полігону

Контейнеризація тестового полігону реалізована через Docker, що забезпечує ізоляцію мікросервісів і їхніх баз даних, а також однакове середовище на етапах розробки, тестування та розгортання [17]. Кожен мікросервіс (auth_service, product_service, order_service) упаковано в окремий контейнер, який взаємодіє з власною базою даних PostgreSQL, також розгорнутою в контейнері.

Як я вже встиг повторити неодноразово у попередніх пунктах, кожен мікросервіс використовує однаковий Dockerfile, базований на образі `python:3.11-slim`. Структура Dockerfile включає:

- копіювання файлу `requirements.txt` і встановлення залежностей (FastAPI, SQLAlchemy, aiohttp, slowapi тощо) через `pip`;
- копіювання коду сервісу в контейнер;
- запуск FastAPI-додатка через Uvicorn на відповідному порті (8000 для `auth_service`, 8002 для `product_service`, 8004 для `order_service`).

Для координації мікросервісів і баз даних використано Docker Compose, який визначає шість служб: `auth_service`, `auth_db`, `product_service`, `product_db`, `order_service`, `order_db`. Ключові аспекти конфігурації наступні.

- служби баз даних: Використовують образ `postgres:latest` із змінними середовища для налаштування користувача, пароля та назви бази (`auth_db`, `product_db`, `order_db`). Порти мапуються на хост (5432, 5433, 5434), а дані зберігаються у томах (`auth_db_data`, `product_db_data`, `order_db_data`);
- служби мікросервісів: Будуються з відповідних директорій (`./auth_service`, `./product_service`, `./order_service`) через Dockerfile. Кожен сервіс має унікальний порт (8000, 8002, 8004) і залежить від своєї бази даних із умовою `service_healthy`, що гарантує готовність PostgreSQL;
- мережа: Усі служби підключено до мостової мережі `app-network`, що забезпечує ізольовану комунікацію;
- залежності: `order_service` додатково залежить від `auth_service` і `product_service` (умова `service_started`), відображаючи їхню взаємодію.

Пропоную тепер трохи детальніше розглянути процес розгортання, який виконується у кілька кроків. У кореневій директорії проєкту, де розташовано `docker-compose.yml` і директорії мікросервісів (`auth_service`, `product_service`, `order_service`), виконується команда `docker-compose up --build`, що будує усі образи (якщо вони ще не створені), створює мережу `app-network` і томи, а також забезпечує послідовний запуск (спочатку бази даних, потім

мікросервіси). Доступність сервісів перевіряється через запити до ендпоінтів GET /health (наприклад, `curl http://localhost:8000/health`). Логи контейнерів можна переглянути командою `docker-compose logs`.

4.4 Структура проєкту тестового полігону

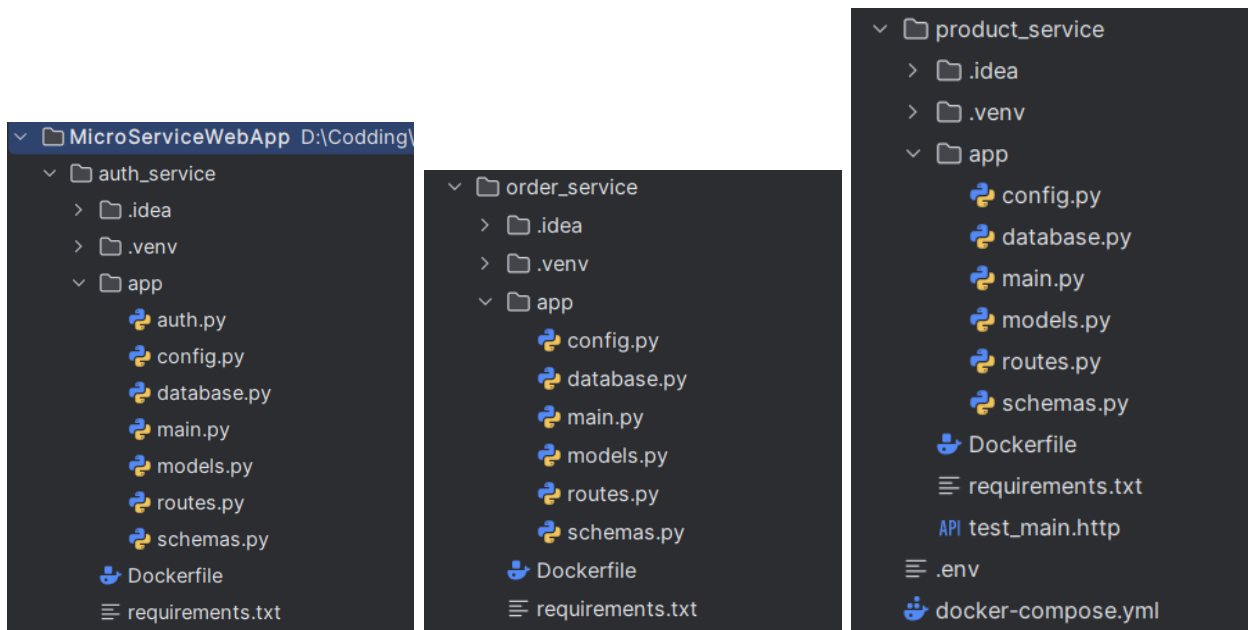
Проєкт тестового полігону організовано як набір незалежних мікросервісів, кожен із яких має власну директорію та ізольовану базу даних. Така модульна структура сприяє чіткому поділу відповідальностей і полегшує розробку, тестування та розгортання [20].

Загальна структура проєкту включає три основні директорії: `auth_service/`, що містить код для автентифікації та управління користувачами; `product_service/`, що відповідає за управління каталогом продуктів; а також `order_service/`, який реалізує логіку створення та перегляду замовлень.

Як вже можна було побачити з попередніх пунктів, кожна директорія мікросервісу має однакову внутрішню структуру:

- `app/`: Основна директорія з кодом;
 - `main.py`: Точка входу FastAPI-додатка, ініціалізація маршрутів і middleware;
 - `models.py`: SQLAlchemy-моделі для бази даних (наприклад, `User` і `Token` для `auth_service`, `Product` для `product_service`, `Order` і `OrderItem` для `order_service`);
 - `schemas.py`: Pydantic-схеми для валідації вхідних і вихідних даних;
 - `config.py`: Клас конфігурації (`AuthConfig`, `ProductConfig`, `OrderConfig`) для управління параметрами;
 - `database.py`: Налаштування підключення до PostgreSQL через SQLAlchemy;
 - `routes.py`: Визначення API-ендпоінтів;
- `Dockerfile`: Описує побудову контейнера для мікросервісу;
- `requirements.txt`: Список залежностей Python.

Кожен мікросервіс використовує власну базу даних (auth_db, product_db, order_db), що забезпечує ізоляцію даних і незалежність сервісів. Наприклад, auth_service працює з таблицею users, product_service – з таблицею products, а order_service – з таблицями orders і order_items. Модульність коду підтримується через чіткий поділ на моделі, схеми та маршрути, що полегшує підтримку та розширення функціоналу (див. рис. 4.4-4.6).



Рисунки 4.4-4.6 – структура пмікросервісного тестового полігону

4.5 Реалізація системи АУКМ

Система АУКМ, що реалізована як cbr_config_service, є центральним компонентом тестового полігону, що забезпечує моніторинг, аналіз і автоматичне налаштування мікросервісів (auth_service, product_service, order_service). Її основна мета – оптимізувати роботу сервісів шляхом вибору конфігурацій на основі прецедентів і поточного стану системи.

4.5.1 Основний функціонал

Система виконує три ключові задачі: моніторинг стану мікросервісів, аналіз прецедентів через CBR-методи та автоматичне оновлення конфігурацій. Моніторинг реалізовано через періодичний збір метрик (навантаження на процесор, використання пам'яті, диска, мережі, час відповіді, доступність) за

допомогою ендпоінтів GET /health кожного сервісу, а також бібліотеки psutil для системних ресурсів [27]. Зібрані дані зберігаються в таблиці cases бази даних PostgreSQL (models.py), яка містить метрики, конфігурації та показники ефективності (performance_metric, adaptation_time).

Конфігурування мікросервісів здійснюється через асинхронні HTTP-запити до ендпоінтів POST /config/update, що дозволяють оновлювати параметри, такі як db_pool_size чи response_delay. Для авторизації АУКМ використовує JWT-токени, отримані від auth_service через POST /login (auth.py). WebSocket-інтерфейс (/ws) забезпечує реальний час оновлення даних і автоматичне конфігурування з заданим інтервалом.

Основні маршрути АУКМ, реалізовані в main.py, включають:

- POST /login – авторизує АУКМ в auth_service, отримуючи JWT-токен;
- GET /configs – повертає поточні конфігурації всіх мікросервісів;
- GET /auto-config – запускає автоматичне конфігурування за вибраним CBR-методом;
- WebSocket /ws – передає метрики та конфігурації в реальному часі, підтримує автоконфігурування;
- PUT /update-config/{service} – оновлює конфігурацію конкретного сервісу;
- GET / – повертає веб-інтерфейс (static/index.html) для моніторингу.

4.5.2 CBR-методи

Система АУКМ використовує п'ять CBR-методів, реалізованих у модулях cbr/*.py, для вибору оптимальних конфігурацій на основі історичних прецедентів:

- KNN (K-Nearest Neighbors): Порівнює поточний стан системи з історичними випадками за евклідовою відстанню, вибираючи k найближчих із мінімальним performance_metric (nearest_neighbor.py) [23];

- Weighted KNN: Розширює KNN, застосовуючи ваги до метрик (наприклад, більша вага для `cpu_load`), що дозволяє враховувати їхню відносну важливість;
- Feature-Based: Обмежує порівняння вибраними ознаками (наприклад, `cpu_load`, `memory_usage`), зменшуючи обчислювальну складність (`feature_based_retrieval.py`);
- Cluster-Based: Використовує K-Means для кластеризації прецедентів і вибирає конфігурацію з найближчого кластера з найкращим `performance_metric` (`cluster_based_retrieval.py`);
- Indexing and Hashing: Застосовує локально-чутливе хешування (LSH) для швидкого пошуку схожих випадків, оптимізуючи час обробки великих наборів даних (`indexing_hashing.py`).

Кожен метод повертає конфігурацію, яка застосовується до мікросервісів через `POST /config/update`. Методи протестовано на основі метрик, таких як `adaptation_time` і `performance_metric`, що забезпечує адаптивність АУКМ до змін у системі [2].

4.6 Контейнеризація та структура системи АУКМ

Про контейнеризацію багато нового не скажу – вона така сама, як і у всіх інших мікросервісів (оскільки сама по собі система АУКМ є таким самим мікросервісом, як і мікросервіси в тестовому полігоні).

Загалом структура системи АУКМ в себе включає (див. рис. 4.7):

- `app/`: Основна директорія з кодом;
 - `main.py`: Точка входу FastAPI, ініціалізація маршрутів і WebSocket;
 - `models.py`: SQLAlchemy-модель Case для зберігання прецедентів;
 - `schemas.py`: Pydantic-схеми для валідації даних;
 - `auth.py`: Логіка авторизації через `auth_service`;
 - `database.py`: Налаштування підключення до PostgreSQL.

- `system_monitor.py`: Збір системних метрик через `psutil` і HTTP-запити;
- `cbr/`: Модулі CBR-методів (`nearest_neighbor.py`, `feature_based_retrieval.py`, тощо);
- `static/`: Містить `index.html` — веб-інтерфейс для моніторингу;
- `logs/`: Зберігає логи сервісу (`service.log`);
- `Dockerfile`: Описує побудову контейнера;
- `requirements.txt`: Залежності (`scikit-learn`, `FastAPI`, `psutil`, тощо).

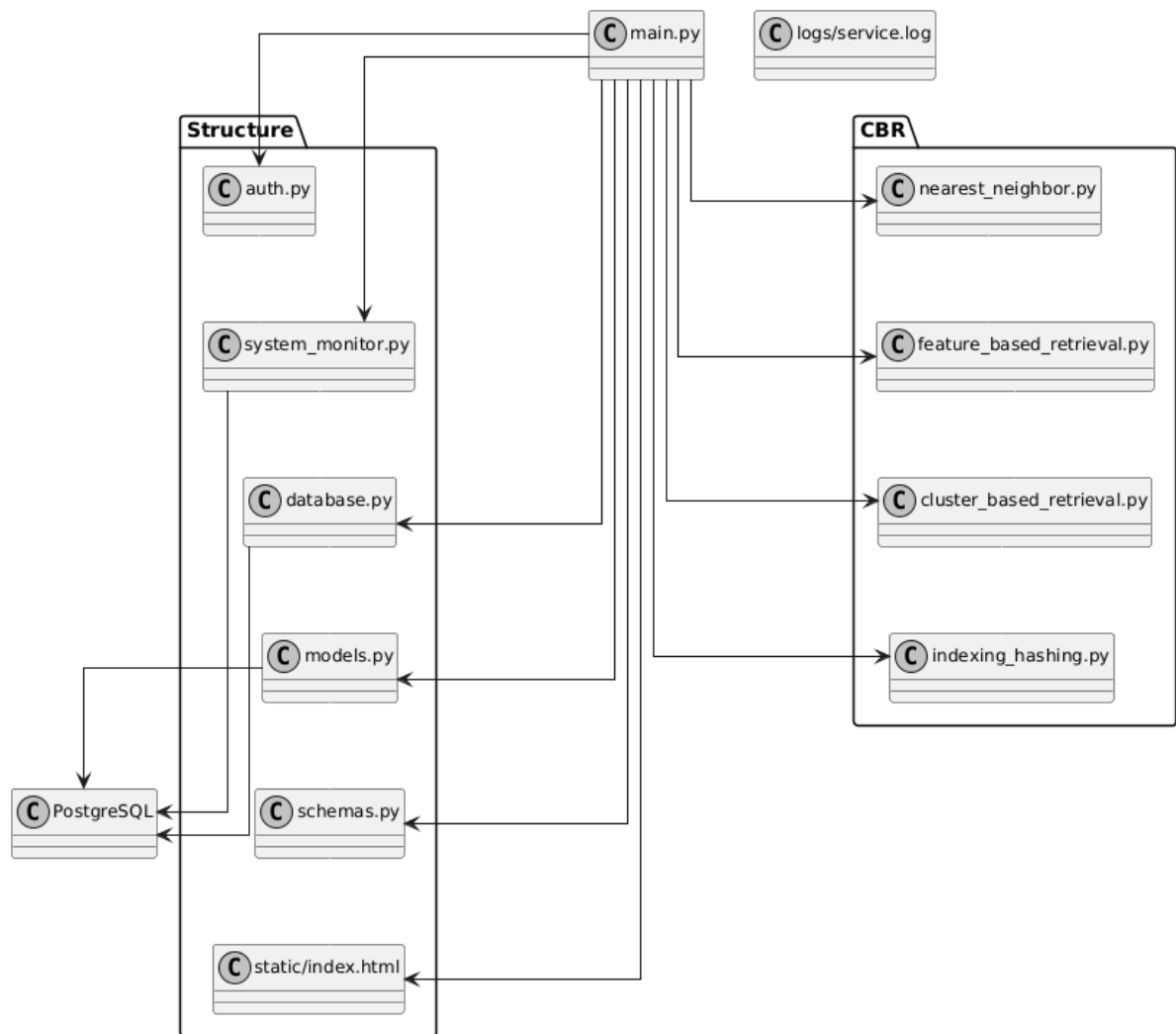


Рисунок 4.7 – Структура АУКМ

Модульна структура дозволяє ізолювати логіку CBR-методів (`cbr/*`), моніторингу (`system_monitor.py`) та авторизації (`auth.py`). Використання PostgreSQL (`cbr_db`) із таблицею `cases` забезпечує збереження прецедентів, а логування в директорії `logs/` - відстеження роботи системи.

Нижче можна побачити взаємозв'язок системи АУКМ із нашим тестовим полігоном на рисунку 4.8.

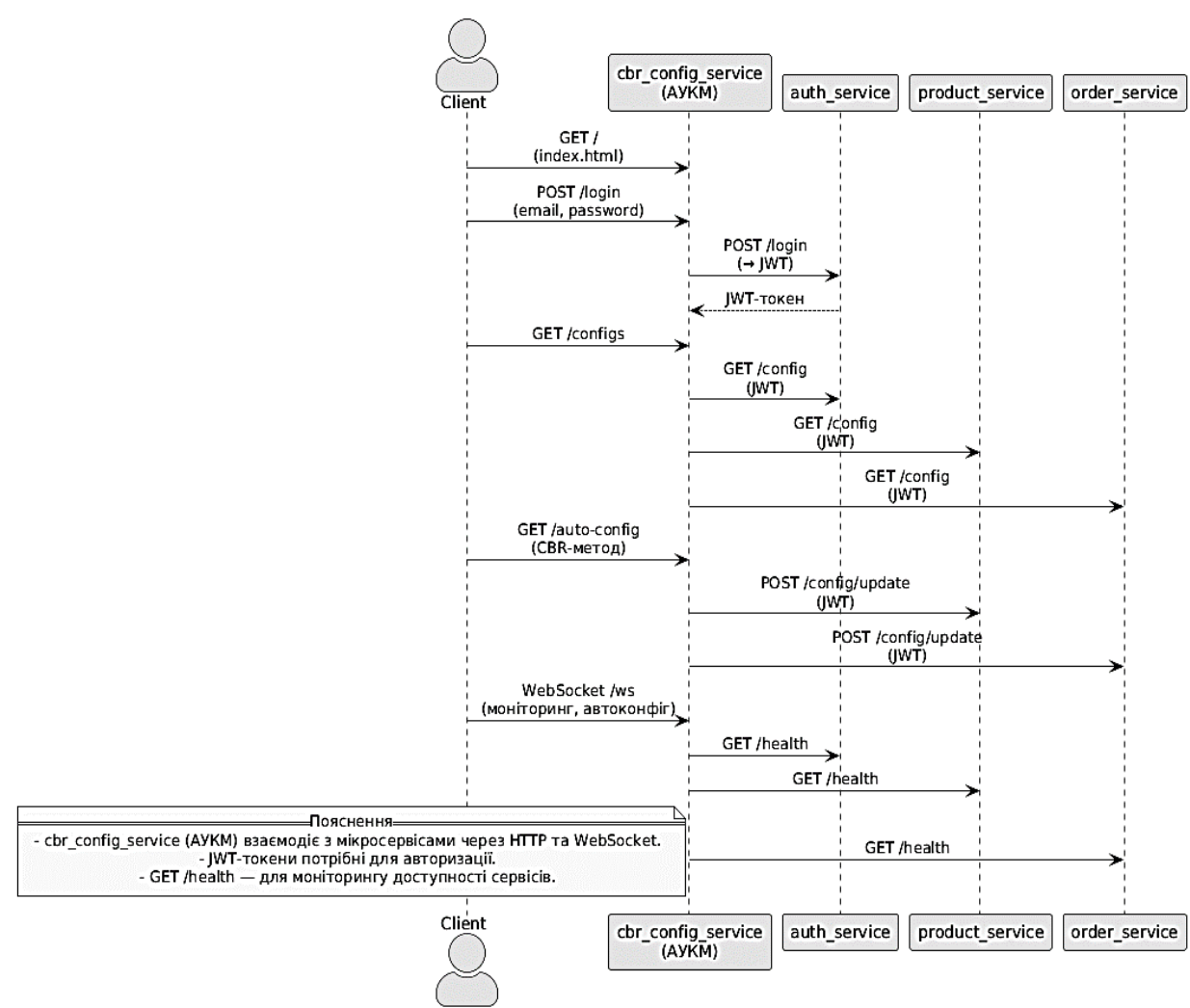


Рисунок 4.8 – Взаємодія АУКМ з тестовим полігоном

4.7 Результат розробки полігону та його тестування

Перш ніж перейти до тестування основної частини моєї роботи, а саме інтелектуальної системи АУКМ, важливо перевірити сам тестовий полігон.

4.7.1 Результат та тестування auth_service

Із структурою та функціоналом auth_service ми вже встигли ознайомитись, тепер настав час наочно побачити, що він собою являє. На рисунках 4.9, 4.10 та 4.11 можна побачити, як виглядає цей сервіс у Swagger UI (його маршрути, а також основні схеми).

Authorize

default	^
GET /health Health Check	▼
POST /config/update Update Config	▼
GET /config Get Config	▼
POST /register Register	▼
POST /login Login	▼
POST /promote/{user_id} Promote User	▼
GET /users Get All Users	▼
GET /me Read Users Me	▼
POST /logout Logout	▼

Рисунок 4.9 – Маршрути auth_service

Body_login_login_post ^ Collapse all object

grant_type > Expand all (string | null)

username* string

password* string

scope > Expand all string

client_id > Expand all (string | null)

client_secret > Expand all (string | null)

ConfigUpdate ^ Collapse all object

access_token_expire_minutes > Expand all (integer | null)

algorithm > Expand all (string | null)

db_connection_type > Expand all (string | null)

db_pool_size > Expand all (integer | null)

rate_limit_requests > Expand all (integer | null)

response_delay > Expand all (number | null)

HTTPValidationError ^ Collapse all object

detail > Expand all array<object>

Token ^ Collapse all object

access_token* string

token_type* string

UserCreate ^ Collapse all object

username* string ≥ 3 characters

email* string email

password* string ≥ 8 characters matches ^[A-Za-z0-9]+\$

UserOut ^ Collapse all object

id* integer

username* string

email* string email

role* string

ValidationError ^ Collapse all object

loc > Expand all array<(string | integer)>

msg* string

type* string

Рисунки 4.10-4.11 – Схеми auth_service

З рисунків 4.10 та 4.11 можна побачити, наприклад, що для створення користувача є певні обмеження – довжина імені користувача має бути більшою за 3 символи, email має відповідати стандарту email, а пароль має містити в собі не менше 8 символів, які включають в себе латинські літери всіх регістрів та цифри від 0 до 9.

Протестуємо основні маршрути додатку. Для цього створимо двох тестових користувачів – один з них стане адміністратором, другий буде рядовим користувачем. Перевіримо основний функціонал для адміністратора

на обох користувачах, аби впевнитись у роботі авторизації. Нехай користувач адмін буде мати наступні дані при реєстрації: username – AdminUser, email – admin@gmail.com, password – admin123. Рядовий користувач буде мати, в свою чергу, наступні дані для входу: username – User, email – user@gmail.com, password – user1234 (див. рис. 4.12-4.13).

Request URL		Request URL	
http://localhost:8000/register		http://localhost:8000/register	
Server response		Server response	
Code	Details	Code	Details
200	Response body <pre>{ "id": 3, "username": "AdminUser", "email": "admin@gmail.com", "role": "user" }</pre> Response headers <pre>access-control-allow-credentials: true access-control-allow-origin: http://localhost:8000 content-length: 71 content-type: application/json date: Wed, 04 Jun 2025 07:26:19 GMT server: uvicorn vary: Origin</pre>	200	Response body <pre>{ "id": 4, "username": "User", "email": "user@gmail.com", "role": "user" }</pre> Response headers <pre>access-control-allow-credentials: true access-control-allow-origin: http://localhost:8000 content-length: 65 content-type: application/json date: Wed, 04 Jun 2025 07:27:06 GMT server: uvicorn vary: Origin</pre>

Рисунки 4.12-4.13 – Результати створення двох нових користувачів

Тепер необхідно зробити користувача AdminUser адміністратором. Для цього є два шляхи. Перший – залогінитись як вже існуючий користувач адміністратор та підвищити роль для цього користувача, або зробити це напямучу через БД. Оберемо другий варіант, аби також встигнути побачити, як виглядає БД для цього сервісу. Для цього, спершу, необхідно підключитись до контейнеру наступною командою: `docker exec -it microservicewebapp-auth_db-1 psql -U postgres -d auth_db`, а після виконати SQL-запит: `SELECT * FROM users;`, і лиш після цього ми можемо побачити вміст таблиці users (див. рис. 4.14).

```
PS D:\Coding\Di\ploma\MicroServiceWebApp> docker exec -it microservicewebapp-auth_db-1 psql -U postgres -d auth_db
psql (17.5 (Debian 17.5-1.pgdg120+1))
Type "help" for help.

auth_db=# SELECT * FROM users;
 id | username | email | password | role
-----+-----+-----+-----+-----
  1 | bogdan | 1@example.com | $2b$12$rP40YwDRDPIv3VCsftotWeL0qUJB/v.YQP0LGMAxwMKQ6UgRIRL6 | ADMIN
  2 | dennyK | 2@example.com | $2b$12$ovj8YJVcgp2dE3HZRR6o9.0GmXxDj1Xnhh0DDTJUeqbYVzmIFH.UW | USER
  3 | AdminUser | admin@gmail.com | $2b$12$pwczQn1ZPnuY0aPi9gNCrei.Zc0yAy0XD9fU.A4y0U7yhU0BKj3Nm | USER
  4 | User | user@gmail.com | $2b$12$TLQMD/LSyVDkM6jiUBi8b.6LEJGdNYTtubn9mhqLw9ys6XFKPIrBi | USER
(4 rows)
```

Рисунок 4.14 – Вміст таблиці Users

Як бачимо, наш користувач AdminUser має ID = 3, тобто нам треба підвищити користувача з цим значенням ID наступною командою: `UPDATE users SET role = 'ADMIN' WHERE id = 3;`. Після цього побачимо наступну картину (див. рис. 4.15).

```
auth_db=# UPDATE users SET role = 'ADMIN' WHERE id = 3;
UPDATE 1
auth_db=# SELECT * FROM users;
 id | username | email | password | role
-----+-----+-----+-----+-----
  1 | bogdan   | 1@example.com | $2b$12$rP40YwDRDPiv3VCsftotWeL0qUJB/v.YQP0L6MgAxwMKQ6UgRIRL6 | ADMIN
  2 | dennyK   | 2@example.com | $2b$12$ovj8YJVcgp2dE3HZRR6o9.0GmXxDj1Xnhh0DDTJUeqbYVzmIFH.UW | USER
  4 | User     | user@gmail.com | $2b$12$TLQMD/LSyVDkM6jiUB18b.GLEJGdNYTtubn9mhqLw9ysGXFKPIrB1 | USER
  3 | AdminUser | admin@gmail.com | $2b$12$pwczQn1ZPnuY0aPi9gNCrei.Zc0yAy0XD9fU.A4y0U7yhU0BKj3Nm | ADMIN
(4 rows)
```

Рисунок 4.15 – Результат оновлення ролі користувача

Тепер можемо авторизуватись через спеціальну кнопку «Authorize» у Swagger UI для того, аби зберегти у cookie токен сесії цього користувача для подальших тестів (див. рис. 4.16).

Available authorizations x

Scopes are used to grant an application different levels of access to data on behalf of the end user. Each API may declare one or more scopes.
API requires the following scopes. Select which ones you want to grant to Swagger UI.

OAuth2PasswordBearer (OAuth2, password)

Authorized

Token URL: login

Flow: password

username: admin@gmail.com

password: *****

Client credentials location: basic

client_secret: *****

Logout

Close

Рисунок 4.16 – Результат авторизації

Тепер можемо перейти до перевірки основних маршрутів, які доступні як адміністратору, так і рядовому користувачу. Перевіримо маршрути GET /me, GET /users, а також GET /config (див. рис. 4.17-4.19).

Request URL	
<code>http://localhost:8000/me</code>	
Server response	
Code	Details
200	Response body <pre>{ "id": 3, "username": "AdminUser", "email": "admin@gmail.com", "role": "admin" }</pre> Response headers <pre>content-length: 72 content-type: application/json date: Wed,04 Jun 2025 07:41:52 GMT server: uvicorn</pre>

Request URL	
<code>http://localhost:8000/config</code>	
Server response	
Code	Details
200	Response body <pre>{ "access_token_expire_minutes": 30, "algorithm": "HS256", "db_connection_type": "sync", "db_pool_size": 5, "rate_limit_requests": 100, "response_delay": 0 }</pre> Response headers <pre>content-length: 146 content-type: application/json date: Wed,04 Jun 2025 07:46:33 GMT server: uvicorn</pre>

Рисунки 4.17-4.18 – Результати виконання запитів GET /me та GET /config від імені адміністратора

Request URL	
<code>http://localhost:8000/users</code>	
Server response	
Code	Details
200	Response body <pre>[{ "id": 1, "username": "bogdan", "email": "1@example.com", "role": "admin" }, { "id": 2, "username": "dennyk", "email": "2@example.com", "role": "user" }, { "id": 4, "username": "User", "email": "user@gmail.com", "role": "user" }, { "id": 3, "username": "AdminUser", "email": "admin@gmail.com", "role": "admin" }]</pre> Response headers <pre>content-length: 275 content-type: application/json date: Wed,04 Jun 2025 07:45:35 GMT server: uvicorn</pre>

Рисунки 4.19 – Результат виконання запиту GET /users від імені адміністратора

Також важливо переглянути, чи дійсно токен сесії зберігся у cookie. Для цього варто перейти у DevTools в браузері та переглянути сховище аплікації (див. рис. 4.20).

Name	Value	Domain ▲	Path	Expires / Max-Age	Size	HttpOnly
JSESSIONID	E60C2387B01005469A185A8BD...	localhost	/	Session	42	✓
access_token	eyJhbGciOiJIUzI1NiIsInR5cCI6I...	localhost	/	2025-06-04T08:11:39.631Z	149	✓

Рисунок 4.20 – Наявність токена сесії у cookie

4.7.2 Результат та тестування product_service

Із структурою та функціоналом product_service ми вже також встигли ознайомитись у попередніх підрозділах, тому тепер знову варто побачити на практиці, як саме він працює. На рисунках 4.21 та 4.22 можна побачити, як виглядає цей сервіс у Swagger UI (його маршрути, а також основні схеми).

The image shows the Swagger UI interface for the product_service. On the left, under the 'default' tab, there is a list of API endpoints with their methods and descriptions:

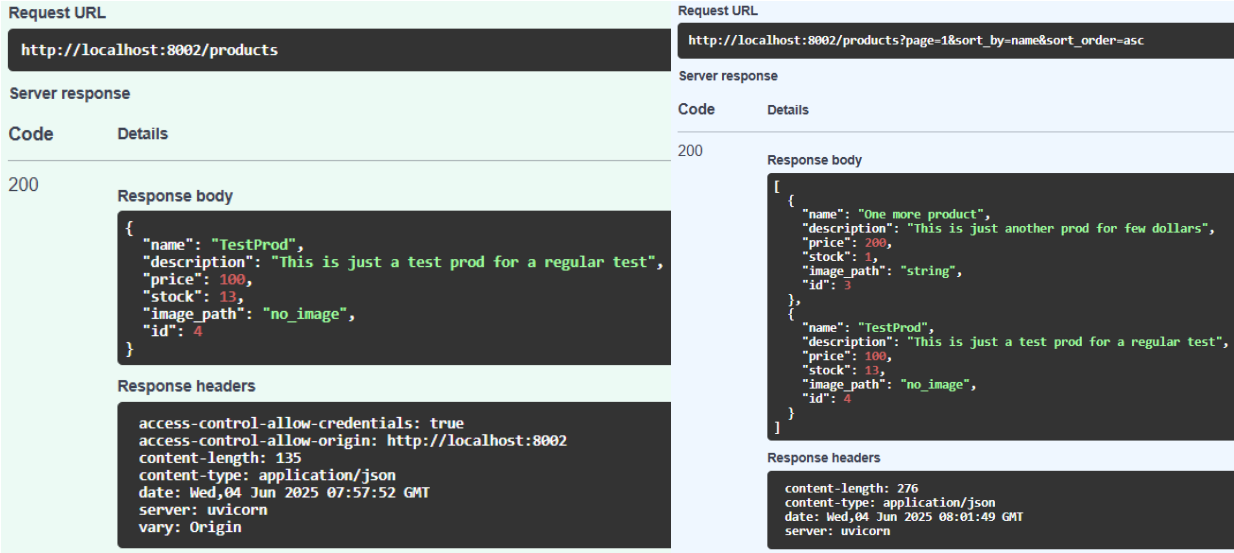
- GET /health** Health Check
- POST /config/update** Update Config
- GET /config** Get Config
- GET /products** Get Products
- POST /products** Create Product
- GET /products/{product_id}** Get Product
- PATCH /products/{product_id}** Update Product Stock
- POST /products/batch** Get Products Batch

On the right, under the 'Schemas' tab, there is a list of data models:

- ConfigUpdate** > Expand all object
- HTTPValidationError** ^ Collapse all object
 - detail** > Expand all array<object>
- ProductCreate** > Expand all object
- ProductOut** > Expand all object
- ValidationError** > Expand all object

Рисунки 4.21-4.22 – Маршрути та схеми product_service

Тестувати, на цей раз, будемо маршрути POST /products та GET /products, аби перевірити повний функціонал створення і перегляду продукту (див. рис. 4.23-4.24). Зазначу, що створення доступне лише для адміністратора, так само як і методи для роботи з конфігом.

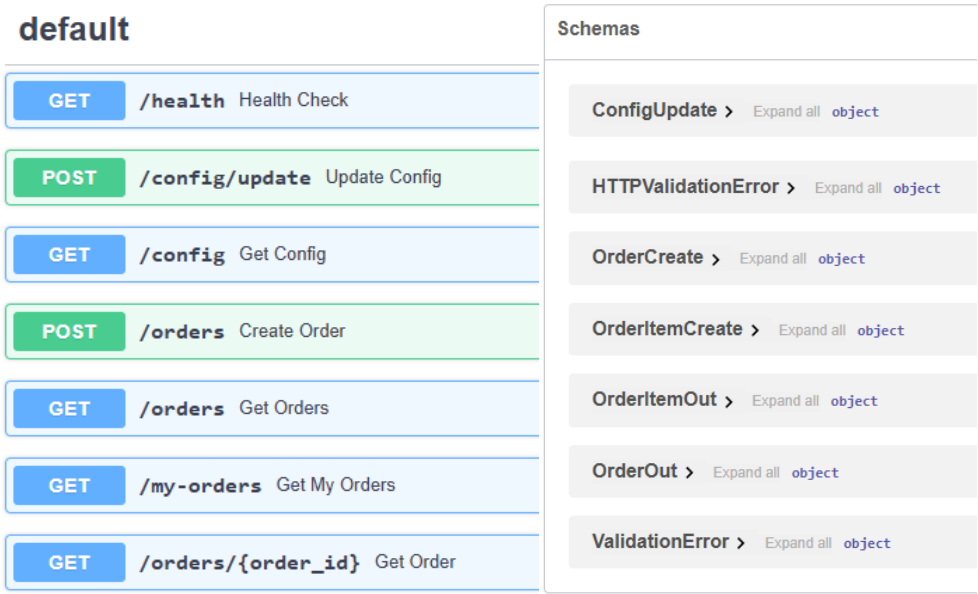


Рисунки 4.23-4.24 – Результат створення та перегляду продуктів

Також, перед тим як перейти до розгляду останнього сервісу у полігоні, хотів би зазначити – кожен з цих сервісів має маршрут GET /health, який використовується пізніше, для моніторингу стану додатку (швидкодії кожного з сервісів, а якщо точніше – перевірка часу відгуку сервісів на запит).

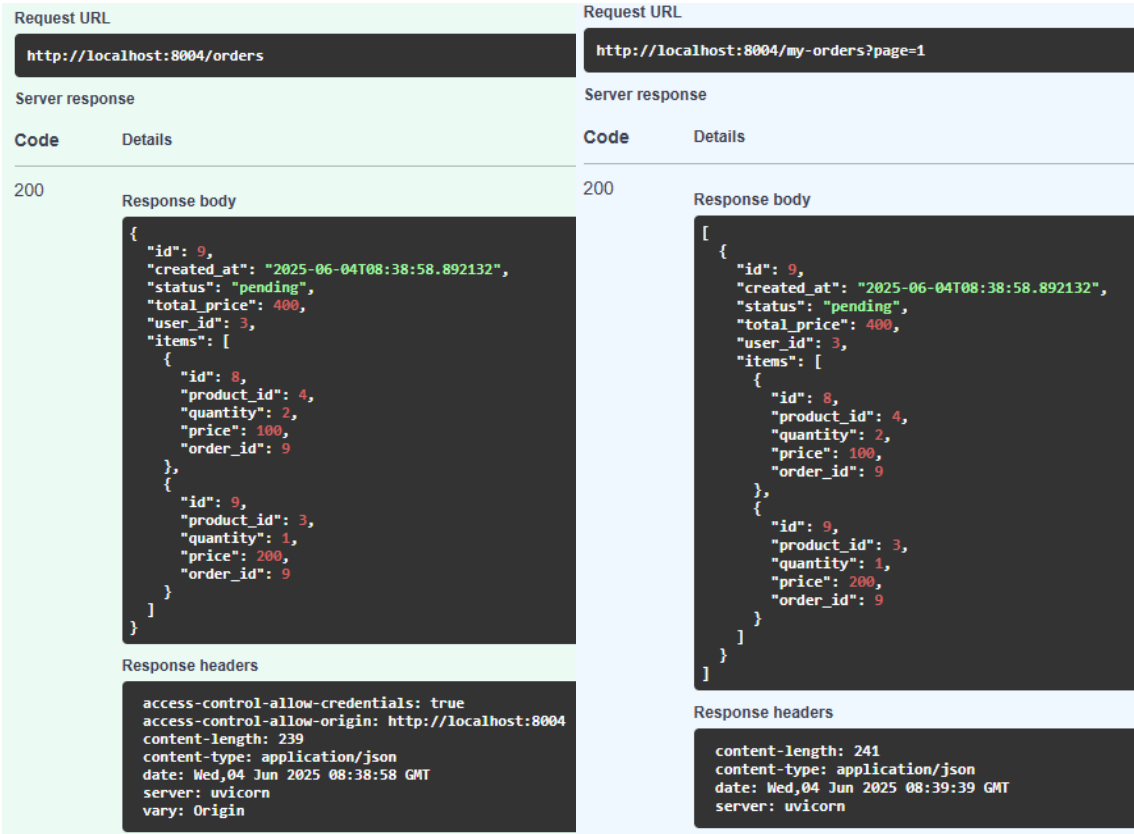
4.7.3 Результат та тестування order_service

Ну і на останок перейдемо до останнього мікросервісу в списку – order_service. Так само спершу маємо ознайомитись із маршрутами та схемами додатку (див. рис. 4.25 та 4.26).



Рисунки 4.25-4.26 – Маршрути та схеми order_service

Тепер перейдемо до тестування основних і головних маршрутів цього сервісу, а саме: POST /orders та GET /my-orders (див. рис. 4.27 та 4.28). Зазначу, що створювати замовлення та переглядати свої замовлення може будь який користувач.



Рисунки 4.27-4.28 – Результат тестування запитів POST /orders та GET /my-orders

Також варто зазначити і показати, що `order_service` дійсно взаємодіє з `product_service`, тому при замовленні певної кількості продуктів, ця кількість віднімається з БД `product_service` (див. рис. 4.29).

```
product_db=# SELECT * FROM products;
 id |      name      | description | price | stock | image_path
----+-----+-----+-----+-----+-----
  4 | TestProd      | This is just a test prod for a regular test | 100 | 11 | no_image
  3 | One more product | This is just another prod for few dollars | 200 | 0 | string
(2 rows)
```

Рисунок 4.29 – Зміна кількості продуктів у БД `product_service`

В свою чергу, запит GET /orders (див. рис. 4.30) доступний тільки для адміністраторів, і дозволяє переглянути усі замовлення, які колись були зроблені.

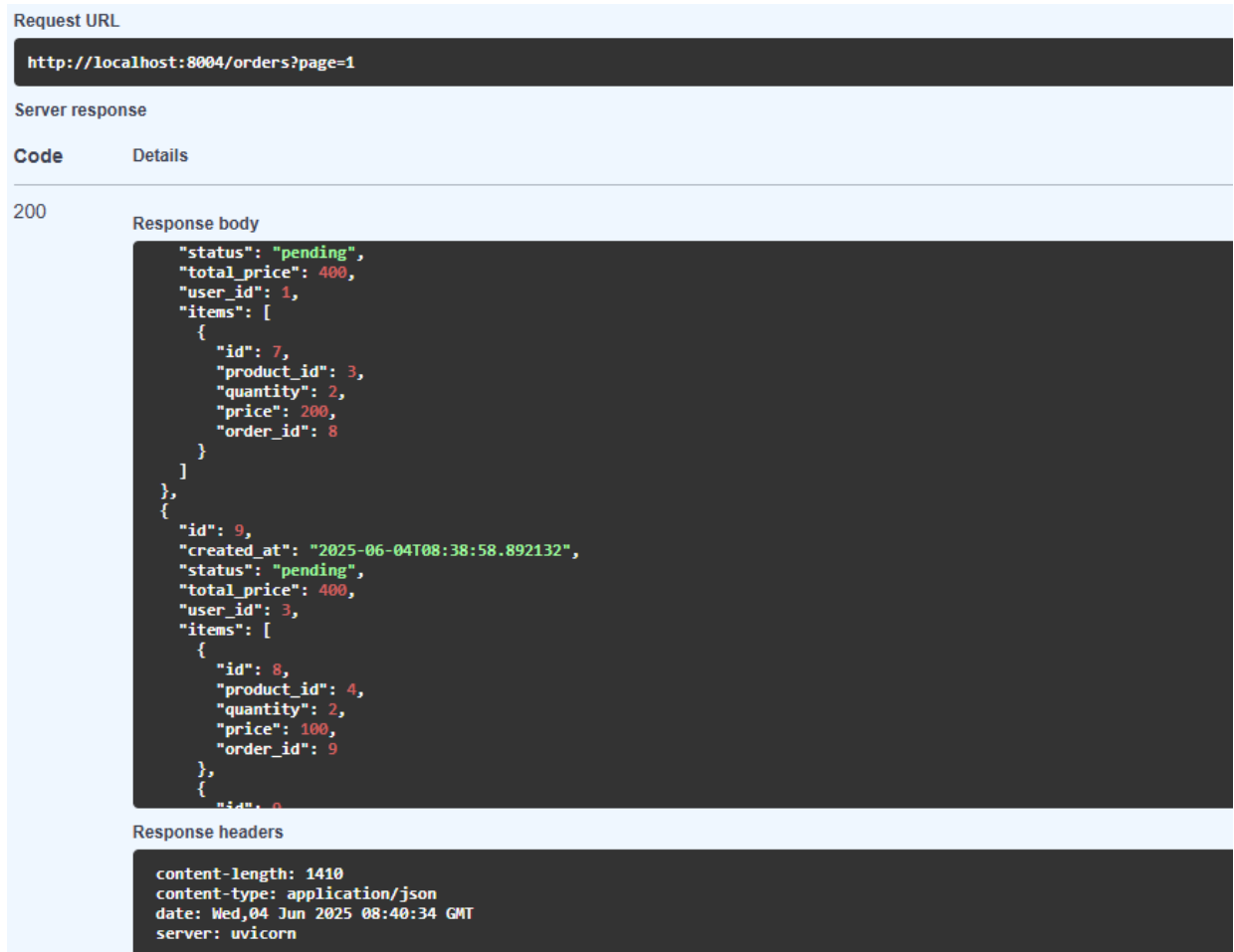


Рисунок 4.30 – Результат виконання запиту GET /orders

Та сама ситуація і з запитами для взаємодії з конфігурацією – доступ є тільки в адміністратора (як на перегляд, так і на оновлення конфігу). Тому, для того, аби мікросервіс для конфігурування мав змогу працювати, нам необхідно буде залогінитись в нього від імені адміністратора, але це розглянемо вже в наступних підрозділах.

4.8 Результат розробки системи АУКМ та її тестування

І ось тепер, нарешті, можемо перейти до найголовнішого. Спершу ознайомимось з інтерфейсом, а далі перейдемо до тестування та дослідів.

4.8.1 Інтерфейс системи АУКМ

Інтерфейс системи АУКМ є односторінковим, поділений на дві вкладки. Перша вкладка дозволяє переглянути поточний стан системи, на якій працює сам полігон, а також вручну, мануально, змінити конфігурації для відповідних мікросервісів.

Перше, що нас зустрічає на цій вкладці – авторизація. Авторизовуватись маємо як адміністратор, оскільки тільки він має права на зміну конфігів тощо (див. рис. 4.31).

CBR Config Dashboard

Admin Login

Email

Password

[Login](#)

Рисунок 4.31 – Авторизація в АУКМ

Після для нас відкривається відразу кілька категорій графіків на відстеження стану системи, будемо рухатись по порядку. Перша категорія – «QoS Metrics» (див. рис. 4.32).

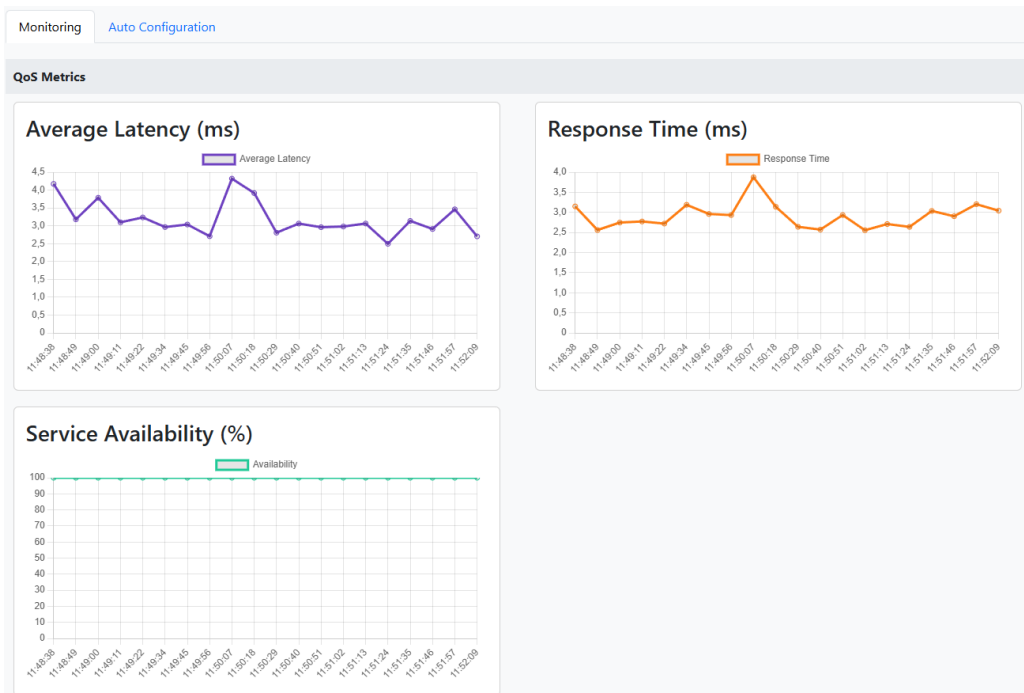


Рисунок 4.32 – Категорія «QoS Metrics»

Це кількісні показники, які використовуються для оцінки якості обслуговування в межах системи. Вони включають в себе: середню затримку (мс), середній час відповіді (мс) та доступність сервісу (у відсотках)

Наступною на черзі є категорія «System Resources» (див. рис. 4.33). В цій категорії визначенні стандартні показники продуктивності системи, а саме: навантаження на процесор (у відсотках), використання оперативної пам'яті (у відсотках), використання диску (у відсотках), а також використання мережі (МБ/с).

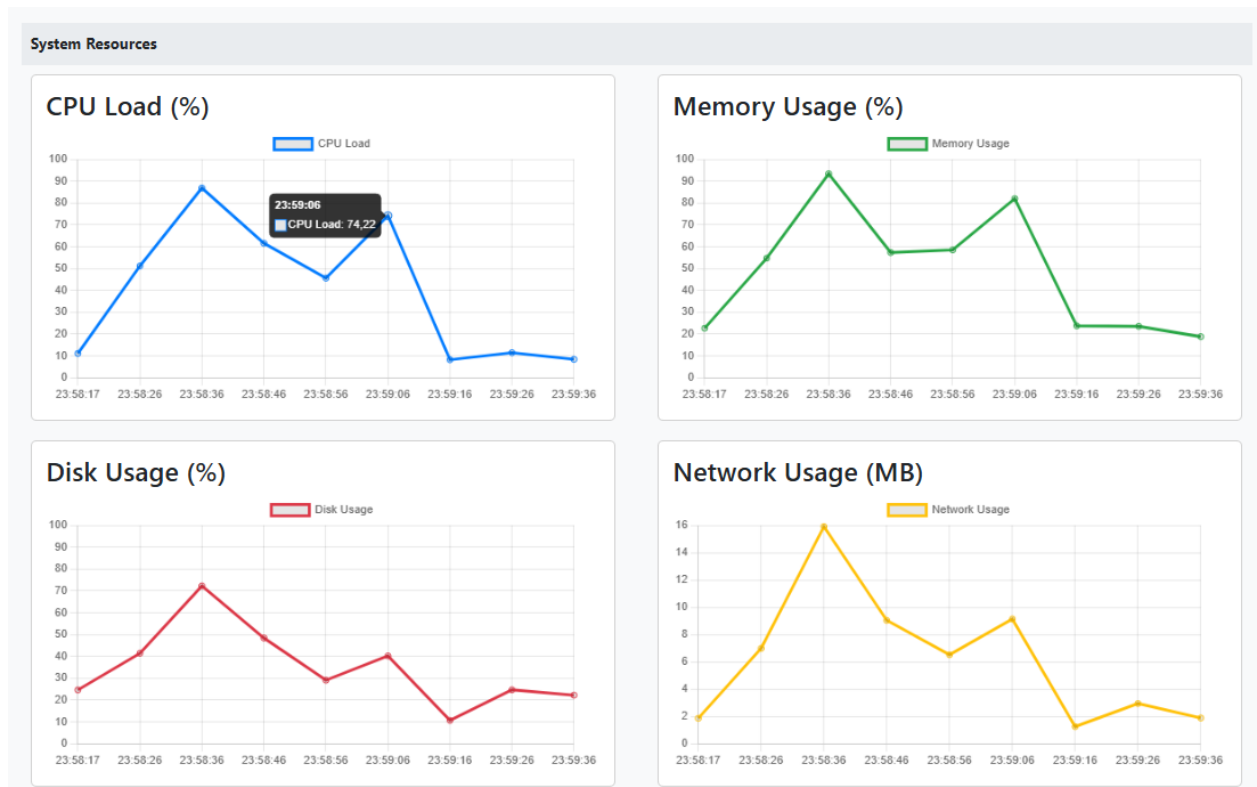


Рисунок 4.33 – Категорія «System Resources»

Далі нас зустрічає категорія «Cost Efficiency Metrics», в якій представлено всього один дашборд – операційні витрати в юнітах (див. рис. 4.34). Цей показник відображає умовні операційні витрати, які пов'язані з роботою системи, тобто допомагає оцінити, наскільки «дорого» обходиться робота системи з точки зору ресурсів. Обчислюється він як зважена сума системних метрик за формулою:

$$OC = CPU * 0.4 + RAM * 0.3 + DISK * 0.2 + NETWORK * 0.1, \quad (4.1)$$

де OC – операційні витрати,

CPU – відсоток використання процесору,

RAM – відсоток використання оперативної пам'яті,

DISK – відсоток використання дискового простору,

NETWORK – обсяг мережевого трафіку в МБ/с.

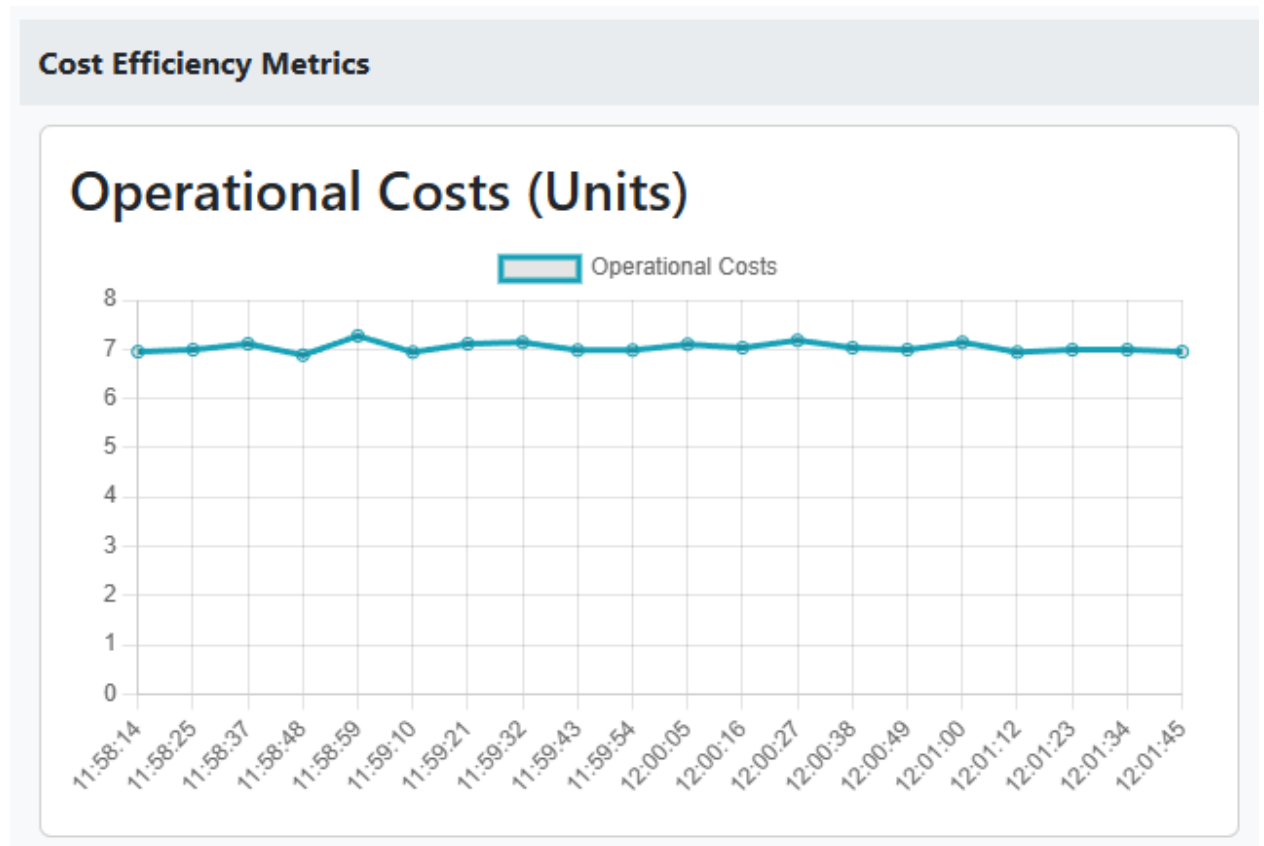


Рисунок 4.34 – Категорія «Cost Efficiency Metrics»

Наступною є категорія дашбордів «Adaptability Metrics» (див. рис. 4.35). В цю категорію входять метрики, які відповідають за адаптивність застосунку – час адаптації, потрібний для автоматичного налаштування конфігурації системи через CBR-метод (в секундах, різниця часу між початком і завершенням виклику GET /auto-config або WebSocket-автоконфігурації), а також реконфігураційність (загальна кількість виконаних реконфігурацій системи).

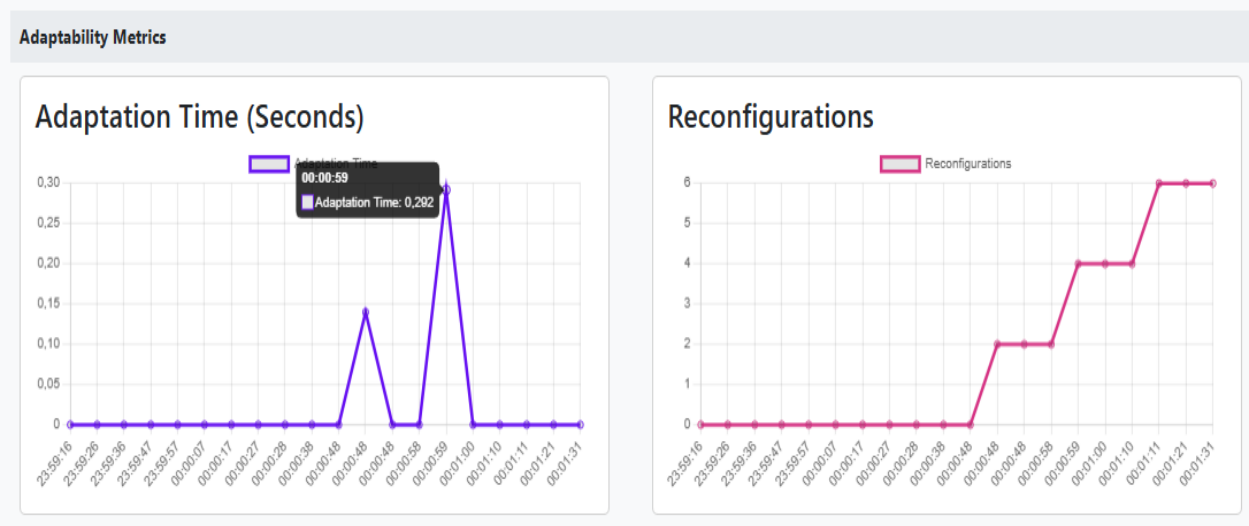


Рисунок 4.35 – Категорія «Adaptability Metrics»

Після всіх дашбордів прописані усі поточні конфігурації для всіх мікросервісів тестового полігону (див. рис. 4.36).

Current Configurations	
Service	Config
auth_service	{ "access_token_expire_minutes": 30, "algorithm": "HS256", "db_connection_type": "sync", "db_pool_size": 5, "rate_limit_requests": 100, "response_delay": 0 }
product_service	{ "db_connection_type": "sync", "db_pool_size": 5, "rate_limit_requests": 100, "response_delay": 0, "max_products_per_page": 20 }
order_service	{ "db_connection_type": "sync", "db_pool_size": 5, "rate_limit_requests": 100, "response_delay": 0, "max_orders_per_page": 20, "default_order_status": "pending" }

Рисунок 4.36 – Поточні конфігурації кожного з мікросервісів

Спробуємо змінити якусь конфігурацію вручну, і подивимось, що станеться. Для цього змінимо час дії токenu у auth_service, а також зменшимо кількість реквестів до 50 (див. рис. 4.37-4.38).

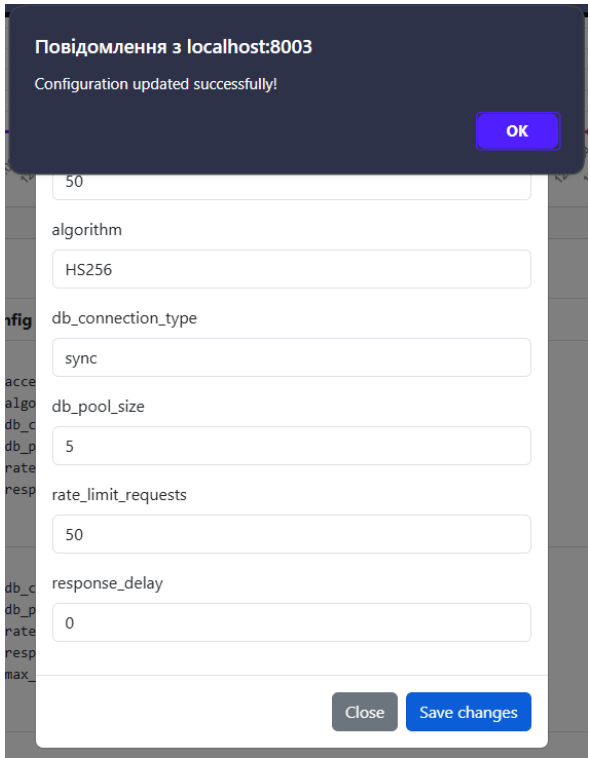


Рисунок 4.37 – Спроба зміни конфігурації

Current Configurations

Service	Config
auth_service	<pre>{ "access_token_expire_minutes": 50, "algorithm": "HS256", "db_connection_type": "sync", "db_pool_size": 5, "rate_limit_requests": 50, "response_delay": 0 }</pre>

Рисунок 4.38 – Результат зміни конфігурації auth_service (успішний)

Наступною на черзі є вкладка Auto Configuration, яка містить в собі всього два розділи – «Automatic Configuration», де ми маємо обрати CBR-метод та можемо включити автоматичну конфігурацію; а також частоту оновлення (в секундах), де ми можемо її змінити (див. рис. 4.39).

При виборі кожного CBR-методу розділ «Automatic Configuration» змінюється відповідно до того, які параметри потребує той чи інший метод (ветод Indexing and Hashing не потребує ніяких параметрів, тому при його виборі категорія лишається без змін).

Monitoring

Auto Configuration

Automatic Configuration

Select CBR Method

K-Nearest Neighbors

☐ Enable Auto-Configuration

Apply Configuration

Update Frequency

Update Interval (seconds, min 5)

10

Apply Interval

Рисунок 4.39 – Вкладка «Auto Configuration»

Як бачимо з рисунку 4.39, метод К-найближчих сусідів не потребує ніяких додаткових параметрів, в той час як інші – потребують. Наприклад – метод Weighted K-Nearest Neighbors потребує обрати «вагу» для кожної з метрики, на основі яких буде встановлюватись конфігурація (див. рис. 4.40).

Automatic Configuration

Select CBR Method

Weighted K-Nearest Neighbors

Weights for Weighted KNN

CPU Load Weight

0,3

Memory Usage Weight

0,3

Disk Usage Weight

0,1

Network Usage Weight

0,1

Average Latency Weight

0,1

Response Time Weight

0,05

Availability Weight

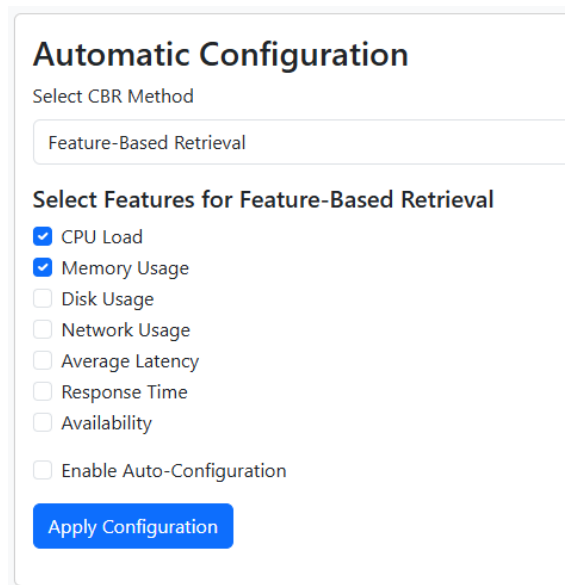
0,05

☐ Enable Auto-Configuration

Apply Configuration

Рисунок 4.40 – Параметри для методу Weighted K-Nearest Neighbors

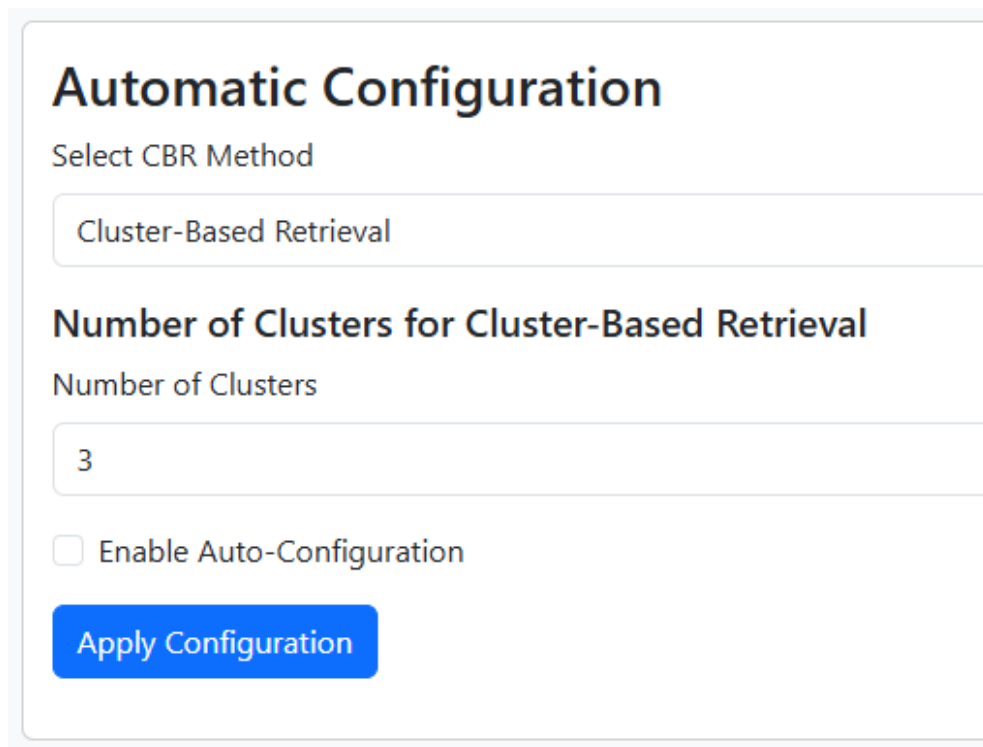
Метод Feature-Based Retrieval включає в себе вибір важливих для нас параметрів, які будуть впливати на вибір конфігурації, тому в ньому вибір параметрів представлений у вигляді чекбоксів (див. рис. 4.41).



The screenshot shows a web form titled "Automatic Configuration". Under the heading "Select CBR Method", the "Feature-Based Retrieval" option is selected in a dropdown menu. Below this, under "Select Features for Feature-Based Retrieval", there are several checkboxes: "CPU Load" and "Memory Usage" are checked, while "Disk Usage", "Network Usage", "Average Latency", "Response Time", "Availability", and "Enable Auto-Configuration" are unchecked. At the bottom of the form is a blue button labeled "Apply Configuration".

Рисунок 4.40 – Параметри для методу Feature-Based Retrieval

Метод Cluster-Based Retrieval потребує тільки одного параметру – кількість кластерів, на які будуть розбиватись усі існуючі прецеденти в БД (див. рис. 4.41).



The screenshot shows a web form titled "Automatic Configuration". Under the heading "Select CBR Method", the "Cluster-Based Retrieval" option is selected in a dropdown menu. Below this, under "Number of Clusters for Cluster-Based Retrieval", there is a text input field labeled "Number of Clusters" containing the value "3". At the bottom of the form, there is an unchecked checkbox for "Enable Auto-Configuration" and a blue button labeled "Apply Configuration".

Рисунок 4.41 – Параметри для методу Cluster-Based Retrieval

Також хотілося б зазначити, що у випадку успішної зміни конфігурації ми отримаємо повідомлення про це в категорії (див. рис. 4.42).

```
Configuration applied successfully!
Method: K-Nearest Neighbors
Parameters: k=3 (default)
Auto-Configuration: Disabled
New config:
{
  "auth_service": {
    "access_token_expire_minutes": 30,
    "algorithm": "HS256",
    "db_connection_type": "sync",
    "db_pool_size": 5,
    "rate_limit_requests": 100,
    "response_delay": 0
  },
  "product_service": {
    "db_connection_type": "sync",
    "db_pool_size": 5,
    "rate_limit_requests": 100,
    "response_delay": 0,
    "max_products_per_page": 20
  },
  "order_service": {
    "db_connection_type": "sync",
    "db_pool_size": 5,
    "rate_limit_requests": 100,
    "response_delay": 0,
    "max_orders_per_page": 20,
    "default_order_status": "pending"
  }
}
```

Рисунок 4.42 – Результат застосування методу K-Nearest Neighbors

На рисунку 4.42 видно, що ми застосували метод K-Nearest Neighbors і система автоматично підбрала для нас конфіг з БД – в даному випадку це стандартний конфіг, оскільки додаток стоїть в простоті і в ньому переважно збережені лише стандартні конфіги. Далі будемо проводити експерименти, в яких конфіги будуть мінятись на щось більш не стандартне.

4.8.2 Проведення експериментів із системою АУКМ

Для того, аби імітувати навантаження на систему було вирішено реалізувати імітацію зміну усіх відповідних параметрів системи через спеціальний скрипт, який генерує значення для кожної з метрики у діапазоні, який відповідає різним типам навантаження, а саме: low_load, medium_load, high_load, peak_load та sub_optimal (таблиця 4.1).

Таблиця 4.1 – Діапазони значень метрик до кожного зі сценаріїв.

Метрика	Low Load	Medium Load	High Load	Peak Load	Suboptimal
cpu_load	5.0-20.0	30.0-50.0	60.0-80.0	80.0-95.0	40.0-70.0
memory_usage	15.0-30.0	40.0-60.0	65.0-85.0	85.0-95.0	50.0-80.0
disk_usage	10.0-25.0	25.0-40.0	40.0-60.0	60.0-75.0	30.0-50.0
network_usage	0.5-3.0 MB/s	3.0-8.0 MB/s	8.0-15.0 MB/s	15.0-25.0 MB/s	5.0-10.0 MB/s
performance_metric	10.0-30.0 ms	30.0-80.0 ms	80.0-150.0 ms	150.0-300.0 ms	200.0-400.0 ms
response_time	5.0-20.0 ms	20.0-50.0 ms	50.0-100.0 ms	100.0-200.0 ms	150.0-300.0 ms
operational_costs	5.0-15.0 units	15.0-30.0 units	30.0-50.0 units	50.0-80.0 units	25.0-60.0 units
availability	95.0-100.0%	90.0-100.0%	85.0-95.0%	80.0-90.0%	70.0-85.0%
adaptation_time	0.1-0.5 s	0.5-1.0 s	1.0-2.0 s	2.0-5.0 s	1.5-3.0 s
reconfigurations	0-2	1-5	3-8	5-10	2-6

Перейдемо до проведення тестування. Першим на черзі буде метод K-Nearest Neighbors. Пропоную відразу переглядати результати в консолі для зручного відображення і конфігу, який було застосовано, і метрик, до яких цей конфіг було застосовано (див. рис. 4.43).

```

2025-06-05 09:21:54,352 - INFO - Starting synthetic status generation
2025-06-05 09:21:54,353 - INFO - Selected scenario: medium_load
2025-06-05 09:21:54,354 - INFO - Completed synthetic status generation
2025-06-05 09:21:54,355 - INFO - System status fetched successfully: {
  "cpu_load": 40.95,
  "memory_usage": 41.9,
  "disk_usage": 38.52,
  "network_usage": 3.47,
  "performance_metric": 50.69,
  "response_time": 49.94,
  "operational_costs": 37.0,
  "availability": 98.18
}
```

Рисунок 4.43 – Результат генерації метрик системи

Як видно з рисунку 4.43 – першим кроком стала генерація поточних метрик системи. Система обрала сценарій середнього завантаження, і на його основі згенерувала відповідні метрики.

На рисунку 4.44 можна побачити результат пошуку найкращого прецеденту для поточних метрик системи.

```
2025-06-05 09:21:54,363 - INFO - Best case selected:
performance_metric=50.36
2025-06-05 09:21:54,363 - INFO - Retrieved config: {
  "auth_service": {
    "access_token_expire_minutes": 60,
    "algorithm": "HS256",
    "db_connection_type": "sync",
    "db_pool_size": 7,
    "rate_limit_requests": 150,
    "response_delay": 0.1
  },
  "product_service": {
    "db_connection_type": "sync",
    "db_pool_size": 7,
    "rate_limit_requests": 150,
    "response_delay": 0.1,
    "max_products_per_page": 30
  },
  "order_service": {
    "db_connection_type": "sync",
    "db_pool_size": 5,
    "rate_limit_requests": 100,
    "response_delay": 0.0,
    "max_orders_per_page": 20,
    "default_order_status": "pending"
  }
}
```

Рисунок 4.44 – Результат пошуку найкращого сценарію

На рисунку 4.45 можна побачити процес застосування конфігурацій.

```
2025-06-05 09:21:54,375 - INFO - Updated configuration for auth_service: {
  "access_token_expire_minutes": 60,
  "algorithm": "HS256",
  "db_connection_type": "sync",
  "db_pool_size": 7,
  "rate_limit_requests": 150,
  "response_delay": 0.1
}
2025-06-05 09:21:54,503 - INFO - Updated configuration for product_service: {
  "db_connection_type": "sync",
  "db_pool_size": 7,
  "rate_limit_requests": 150,
  "response_delay": 0.1,
  "max_products_per_page": 30
}
2025-06-05 09:21:54,618 - INFO - Updated configuration for order_service: {
  "db_connection_type": "sync",
  "db_pool_size": 5,
  "rate_limit_requests": 100,
  "response_delay": 0.0,
  "max_orders_per_page": 20,
  "default_order_status": "pending"
}
```

Рисунок 4.45 – Результат застосування конфігів найкращого сценарію до кожного з мікросервісів

Ну і вже в самому кінці система сповіщає нас, що було обрано наступний конфіг, завдяки певному CBR-методу, із конкретними параметрами для відповідних метрик. В цей час у інтерфейсі застосунок бачимо наступне (див. рис. 4.46).

```
Configuration applied successfully!
Last Updated: 2025-06-05 12:21:54
Method: K-Nearest Neighbors
Parameters: k=3 (default)
Auto-Configuration: Disabled
New config:
{
  "auth_service": {
    "access_token_expire_minutes": 60,
    "algorithm": "HS256",
    "db_connection_type": "sync",
    "db_pool_size": 7,
    "rate_limit_requests": 150,
    "response_delay": 0.1
  },
  "product_service": {
    "db_connection_type": "sync",
    "db_pool_size": 7,
    "rate_limit_requests": 150,
    "response_delay": 0.1,
    "max_products_per_page": 30
  },
  "order_service": {
    "db_connection_type": "sync",
    "db_pool_size": 5,
    "rate_limit_requests": 100,
    "response_delay": 0,
    "max_orders_per_page": 20,
    "default_order_status": "pending"
  }
}
```

Рисунок 4.46 – Результат вдалого застосування конфігу до системи з поточними метриками

Також варто зазначити, що для заповнення таблиці прецедентів було розроблено скрипт (див. табл. 4.1), який заповнює таблицю 50 різними кейсами, кожен з яких генерується відповідно до сценарію навантаження. Результати показують, що CBR-метод KNN працює коректно. Далі були проведені аналогічні тести для інших CBR-методів.

На рисунку 4.47 показані результати застосування конфігурацій методом Weighted KNN.

```

2025-06-05 10:06:08,683 - INFO - Configuration applied successfully!
Last Updated: 2025-06-05 10:06:08
Method: Weighted Knn
Weights: {
  "cpu_load": 0.3,
  "memory_usage": 0.3,
  "disk_usage": 0.1,
  "network_usage": 0.1,
  "performance_metric": 0.1,
  "response_time": 0.05,
  "availability": 0.05
}
Auto-Configuration: Enabled
Update Interval: 30 seconds
New config:
{
  "auth_service": {
    "access_token_expire_minutes": 30,
    "algorithm": "HS256",
    "db_connection_type": "sync",
    "db_pool_size": 5,
    "rate_limit_requests": 100,
    "response_delay": 0.0
  },
  "product_service": {
    "db_connection_type": "sync",
    "db_pool_size": 7,
    "rate_limit_requests": 150,
    "response_delay": 0.1,
    "max_products_per_page": 30
  },
  "order_service": {
    "db_connection_type": "sync",
    "db_pool_size": 8,
    "rate_limit_requests": 150,
    "response_delay": 0.1,
    "max_orders_per_page": 30,
    "default_order_status": "pending"
  }
}
Applied for the following metrics (System state: medium_load):
{
  "cpu_load": 36.75,
  "memory_usage": 50.02,
  "disk_usage": 26.8,
  "network_usage": 3.42,
  "performance_metric": 45.2,
  "response_time": 47.11,
  "operational_costs": 35.41,
  "availability": 97.46
}

```

Рисунок 4.47 – Результат застосування конфігів методом Weighted K-Nearest Neighbors

З рисунку 4.47 видно, що система ідентифікована як `medium_load` з метриками: `cpu_load` 36.75%, `memory_usage` 50.02%, `disk_usage` 26.8%, `network_usage` 3.42 MB/s, `performance_metric` 45.2 ms, `response_time` 47.11 ms, `operational_costs` 35.41 units, `availability` 97.46% (лише `operational_costs` перевищує 30.0 units). Застосовано Weighted KNN з вагами: `cpu_load` 0.3,

memory_usage 0.3, інші 0.1-0.05, обравши оптимальні параметри: auth_service (30, 5, 100, 0.0), product_service (7, 150, 0.1, 30), order_service (8, 150, 0.1, 30). Конфігурація відповідає medium_load.

На рисунку 4.48 показані результати застосування конфігурацій методом Feature-Based Retrieval.

```
2025-06-05 10:09:15,005 - INFO - Configuration applied successfully!
Last Updated: 2025-06-05 10:09:15
Method: Feature Based
Features: [
  "cpu_load",
  "network_usage",
  "performance_metric"
]
Auto-Configuration: Enabled
Update Interval: 30 seconds
New config:
{
  "auth_service": {
    "access_token_expire_minutes": 15,
    "algorithm": "HS256",
    "db_connection_type": "sync",
    "db_pool_size": 3,
    "rate_limit_requests": 50,
    "response_delay": 0.5
  },
  "product_service": {
    "db_connection_type": "sync",
    "db_pool_size": 3,
    "rate_limit_requests": 50,
    "response_delay": 0.5,
    "max_products_per_page": 10
  },
  "order_service": {
    "db_connection_type": "sync",
    "db_pool_size": 4,
    "rate_limit_requests": 80,
    "response_delay": 0.4,
    "max_orders_per_page": 15,
    "default_order_status": "pending"
  }
}
Applied for the following metrics (System state: suboptimal):
{
  "cpu_load": 62.76,
  "memory_usage": 55.85,
  "disk_usage": 41.85,
  "network_usage": 5.07,
  "performance_metric": 366.59,
  "response_time": 279.27,
  "operational_costs": 50.74,
  "availability": 81.74
}
```

Рисунок 4.48 – Результат застосування конфігів методом Feature-Based Retrieval

З рисунку 4.48 можемо побачити, що на цей раз система була ідентифікована як suboptimal з метриками: cpu_load 62.76%, memory_usage

55.85%, disk_usage 41.85%, network_usage 5.07 MB/s, performance_metric 366.59 ms, response_time 279.27 ms, operational_costs 50.74 units, availability 81.74%, усі в межах діапазонів. Застосовано Feature Based з ознаками: cpu_load, network_usage, performance_metric, обравши неоптимальні параметри: auth_service (15, 3, 50, 0.5), product_service (3, 50, 0.5, 10), order_service (4, 80, 0.4, 15). Конфігурація відповідає suboptimal.

На рисунку 4.49 показані результати застосування конфігурацій методом Cluster-Based Retrieval.

```
2025-06-05 10:10:29,463 - INFO - Configuration applied successfully!
Last Updated: 2025-06-05 10:10:29
Method: Cluster Based
Number of Clusters: 5
Auto-Configuration: Enabled
Update Interval: 30 seconds
New config:
{
  "auth_service": {
    "access_token_expire_minutes": 60,
    "algorithm": "HS256",
    "db_connection_type": "sync",
    "db_pool_size": 7,
    "rate_limit_requests": 150,
    "response_delay": 0.1
  },
  "product_service": {
    "db_connection_type": "sync",
    "db_pool_size": 7,
    "rate_limit_requests": 150,
    "response_delay": 0.1,
    "max_products_per_page": 30
  },
  "order_service": {
    "db_connection_type": "sync",
    "db_pool_size": 5,
    "rate_limit_requests": 100,
    "response_delay": 0.0,
    "max_orders_per_page": 20,
    "default_order_status": "pending"
  }
}
Applied for the following metrics (System state: low_load):
{
  "cpu_load": 18.43,
  "memory_usage": 27.62,
  "disk_usage": 19.76,
  "network_usage": 1.49,
  "performance_metric": 19.6,
  "response_time": 8.68,
  "operational_costs": 19.76,
  "availability": 97.99
}
```

Рисунок 4.49 – Результат застосування конфігів методом Cluster-Based Retrieval

В цьому випадку система була ідентифікована як `low_load` з метриками: `cpu_load` 18.43%, `memory_usage` 27.62%, `disk_usage` 19.76%, `network_usage` 1.49 MB/s, `performance_metric` 19.6 ms, `response_time` 8.68 ms, `operational_costs` 19.76 units, `availability` 97.99%, усі в межах діапазонів, зазначених в таблиці 4.1. Застосовано `Cluster Based` з 5 кластерами, обравши оптимальні параметри: `auth_service` (60, 7, 150, 0.1), `product_service` (7, 150, 0.1, 30), `order_service` (5, 100, 0.0, 20). Конфігурація відповідає `low_load`.

На рисунку 4.50 показані результати застосування конфігурацій методом `Indexing and Hashing`.

```
2025-06-05 10:11:29,463 - INFO - Configuration applied successfully!
Last Updated: 2025-06-05 10:11:29
Method: Indexing and Hashing
Auto-Configuration: Enabled
Update Interval: 30 seconds
New config:
{
  "auth_service": {
    "access_token_expire_minutes": 30,
    "algorithm": "HS256",
    "db_connection_type": "sync",
    "db_pool_size": 3,
    "rate_limit_requests": 60,
    "response_delay": 0.5
  },
  "product_service": {
    "db_connection_type": "sync",
    "db_pool_size": 3,
    "rate_limit_requests": 60,
    "response_delay": 0.5,
    "max_products_per_page": 50
  },
  "order_service": {
    "db_connection_type": "sync",
    "db_pool_size": 3,
    "rate_limit_requests": 30,
    "response_delay": 1,
    "max_orders_per_page": 30,
    "default_order_status": "pending"
  }
}
Applied for the following metrics (System state: high_load):
{
  "cpu_load": 72.34,
  "memory_usage": 78.56,
  "disk_usage": 52.12,
  "network_usage": 12.45,
  "performance_metric": 120.89,
  "response_time": 75.67,
  "operational_costs": 42.34,
  "availability": 89.45
}
```

Рисунок 4.50 – Результат застосування конфігів методом `Indexing and Hashing`

У цьому випадку система була ідентифікована як `high_load` з метриками: `cpu_load` 72.34%, `memory_usage` 78.56%, `disk_usage` 52.12%, `network_usage` 12.45 MB/s, `performance_metric` 120.89 ms, `response_time` 75.67 ms, `operational_costs` 42.34 units, `availability` 89.45%, усі в межах діапазонів таблиці 4.1. Застосовано Indexing and Hashing, обравши оптимальні параметри: `auth_service` (30, 3, 60, 0.5), `product_service` (3, 60, 0.5, 50), `order_service` (3, 30, 1, 30). Конфігурація відповідає `high_load`.

Проведені експерименти показали, що кожен із перелічених методів коректно обирає конфігурації, відповідно до сценаріїв навантаження, зазначених у таблиці 4.1.

4.9 Порівняння часу виконання CBR-методів

Порівняння ефективності часу виконання кожного з алгоритмів проводилось в межах полігону при динамічному оновленні метрик. Швидкість вимірювалась залежно від розміру бази прецедентів: 50, 100, 200, 500 та 1000 записів. Для кожного з CBR-методів вимірювання відбувалось по 100 разів, і в якості результату бралось середнє арифметичне. В деяких методах було встановлено специфічні параметри, про це буде зазначено окремо. Тестування відбувалось у системі з наступними характеристиками:

- операційна система: Windows 10 22H2;
- оперативна пам'ять: 16GB, DDR4, 3200MHz;
- центральний процесор: Ryzen 5 4600H, 6 ядр / 12 потоків.

Першим методом на тестуванні став метод KNN із параметром $k=3$. За результатами дослідів середній час виконання алгоритму для 50 записів у БД прецедентів склав 44.8 мс, для 100 записів – 58.3 мс, для 200 записів – 68.9 мс, для 500 записів – 105.2 мс, ну і для 1000 записів – 152.4 мс. Загалом метод продемонстрував лінійне зростання часу із збільшенням кількості даних у таблиці прецедентів (див. рис. 4.51), що пояснюється необхідністю послідовного порівняння всіх прецедентів для обчислення евклідової відстані.

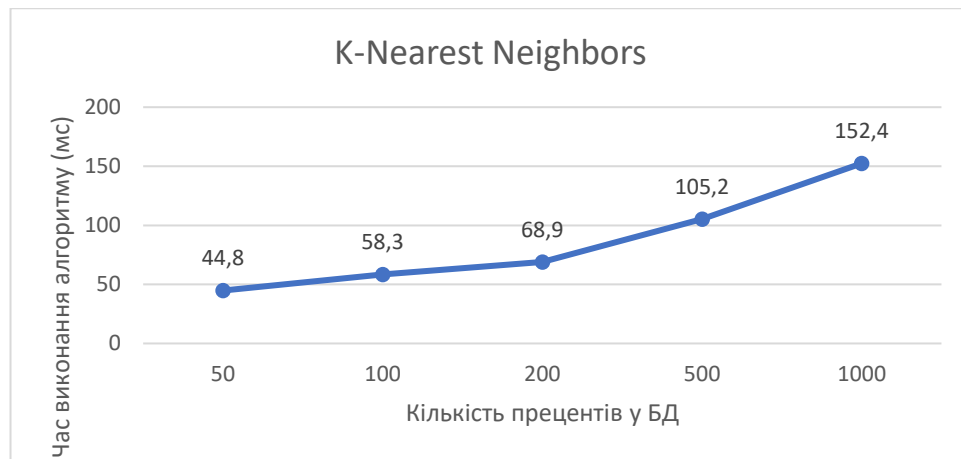


Рисунок 4.51 – Графік залежності часу виконання алгоритму KNN від кількості записів у таблиці прецедентів

Другим методом на тестуванні став Weighted KNN із наступними параметрами ваги: `cpu_load` 0.3, `memory_usage` 0.3, інші від 0.1 до 0.05. Середній час виконання склав: 49.6 мс для 50 записів, 64.1 мс для 100 записів, 75.8 мс для 200 записів, 112.7 мс для 500 записів, і 165.9 мс для 1000 записів. Уповільнення при збільшенні обсягу даних зумовлене додатковими обчисленнями ваг для кожної метрики, що ускладнює процес пошуку (див. рис. 4.52).

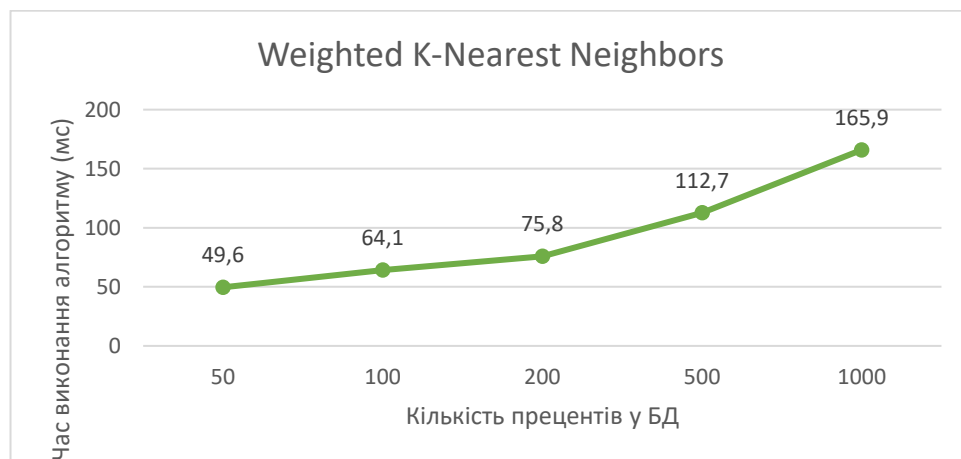


Рисунок 4.52 – Графік залежності часу виконання алгоритму Weighted KNN від кількості записів у таблиці прецедентів

Третім методом в дослідженні виступив Feature-Based Retrieval із обраними ознаками `cpu_load`, `performance_metric`, `response_time` та `availability`. Середній час виконання алгоритму: 47.3 мс для 50 записів, 61.9 мс для 100

записів, 72.5 мс для 200 записів, 109.8 мс для 500 записів та 159.2 мс для 1000 записів. Зростання часу виконання алгоритму пояснюється необхідністю фільтрації та порівняння лише вибраних метрик, що додає додаткові витрати при збільшенні бази (див. рис. 4.53).

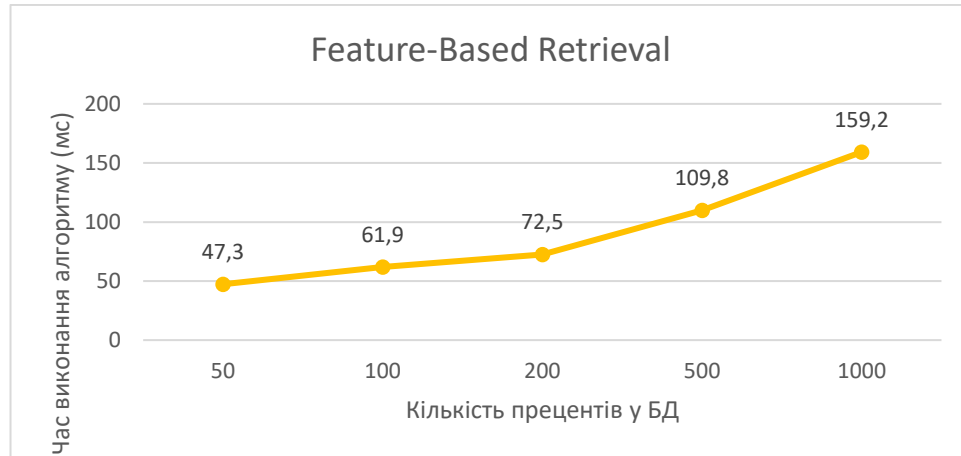


Рисунок 4.53 – Графік залежності часу виконання алгоритму Feature-Based Retrieval від кількості записів у таблиці прецедентів

Четвертим методом в тестах виступив Cluster-Based Retrieval із кількістю кластерів рівною 3. Середній час виконання склав: 53.9 мс для 50 записів, 69.2 мс для 100 записів, 82.4 мс для 200 записів, 126.5 мс для 500 записів та 190.8 мс для 1000 записів. Уповільнення для цього методу зумовлене попередньою кластеризацією, яка вимагає додаткових обчислень для групування прецедентів (див. рис. 4.54).

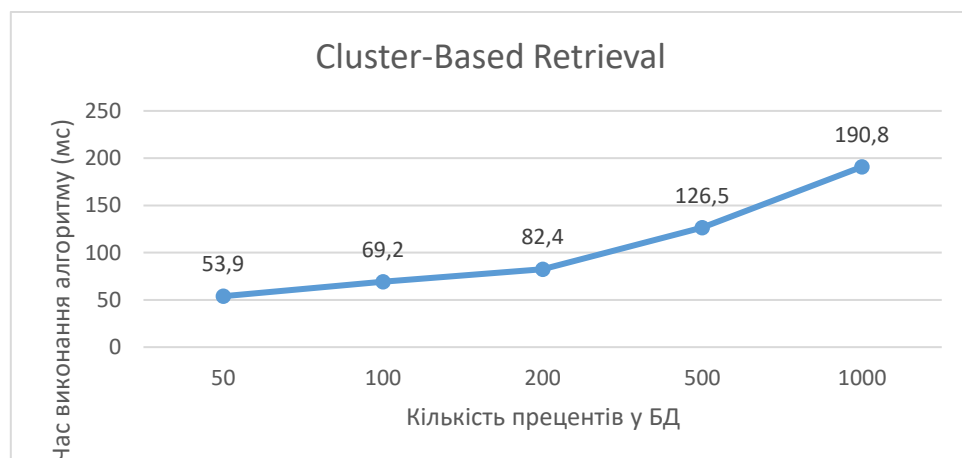


Рисунок 4.54 – Графік залежності часу виконання алгоритму Cluster-Based Retrieval від кількості записів у таблиці прецедентів

П'ятим методом став Indexing and Hashing. Середній час виконання склав: 27.6 мс для 50 записів, 32.1 мс для 100 записів, 36.9 мс для 200 записів, 43.2 мс для 500 записів і 50.3 мс для 1000 записів. Зростання часу виконання цього алгоритму пояснюється розширенням хеш-таблиць, хоча це залишається найшвидшим підходом завдяки попередній індексації (див. рис. 4.55).

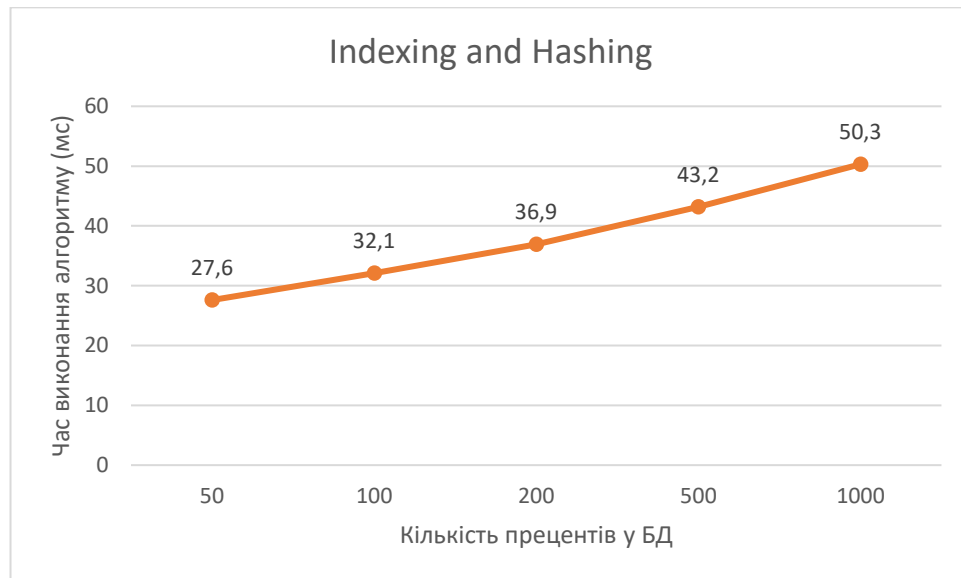


Рисунок 4.55 – Графік залежності часу виконання алгоритму Indexing and Hashing від кількості записів у таблиці прецедентів

Після проведення усіх досліджень стало очевидно, що найшвидшим методом виступив саме Indexing and Hashing, як і очікувалось. Його велика швидкість зумовлена попередньою індексацією записів у таблиці прецедентів. У таблиці 4.2 можна детальніше ознайомитись із результатами дослідження швидкодії кожного з CBR-методів.

Таблиця 4.2 – Порівняння результатів виміру швидкодії CBR-методів.

Метод	Час виконання алгоритмів (мс)				
	для 50 записів	для 100 записів	для 200 записів	для 500 записів	для 1000 записів
K-Nearest Neighbors	44,8	58,3	68,9	105,2	152,4
Weighted K-Nearest Neighbors	49,6	64,1	75,8	112,7	165,9
Feature-Based Retrieval	47,3	61,9	72,5	109,8	159,2
Cluster-Based Retrieval	53,9	69,2	82,4	126,5	190,8
Indexing and Hashing	27,6	32,1	36,9	43,2	50,3

На основі таблиці 4.2 було побудовано порівняльну стовпчасту діаграму (див. рис. 4.56), на якій можна наочно побачити відмінність у швидкості виконання всіх алгоритмів.

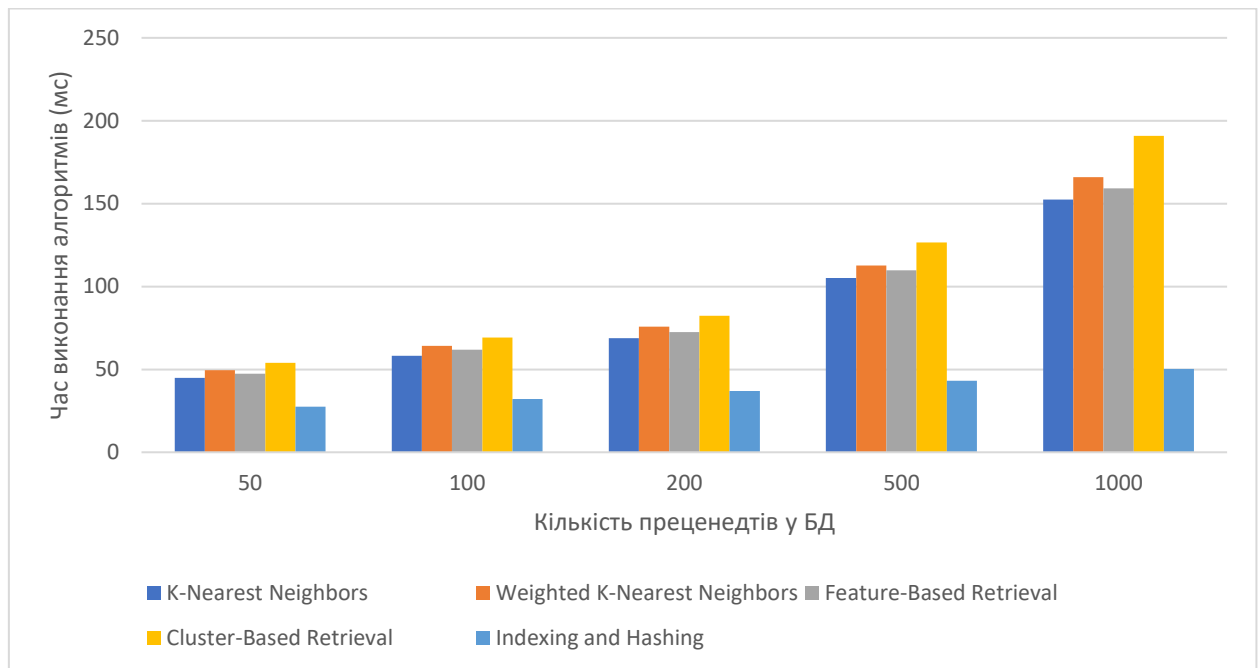


Рисунок 4.56 – Стовпчаста діаграма порівняння швидкодії всіх CBR-методів

Підсумовуючи результати дослідження можна з впевненістю сказати, що тести підтверджують високу швидкість Indexing and Hashing завдяки індексації, що скорочує час пошуку прецедентів [28], хоча точність може знижуватися через спрощене зіставлення [29]. KNN демонструє лінійне зростання часу через послідовне порівняння, але забезпечує стабільність при малих даних [2]. Weighted KNN, попри повільніші вагові обчислення, пропонує гнучкість при пріоритеті метрик, як `cpu_load` [2]. Feature-Based Retrieval і Cluster-Based Retrieval є компромісом між швидкістю та складністю, з додатковими витратами кластеризації та обмеженнями ознак [29, 6].

Вибір методу залежить від пріоритету: Indexing and Hashing для швидкості, Weighted KNN для адаптивності.

ВИСНОВКИ

У рамках виконання моєї кваліфікаційної роботи бакалавра було розроблено систему інтелектуального адаптивного управління конфігураціями мікросервісних застосунків на основі методів аналізу прецедентів.

Було проведено огляд особливостей мікросервісної архітектури, проаналізовані методи та існуючі програмні засоби для управління конфігураціями мікросервісів, а також можливості конфігурування мікросервісів з використанням методів штучного інтелекту.

Було реалізовано п'ять різних CBR-методів. Був спроектований та розроблений тестовий мікросервісний веб-застосунок для використання його в якості тестового полігону.

Був спроектований та розроблений програмний засіб у вигляді окремого мікросервісу для інтелектуального конфігурування мікросервісних застосунків, який включає в себе систему моніторингу та модуль прийняття рішень.

Було проведено експериментальні дослідження можливості використання кожного з досліджуваних методів, які підтвердили коректність роботи модулю прийняття рішень, а також дали змогу оцінити залежність часу виконання алгоритмів різних CBR-методів від об'єму бази прецедентів. Запропоновані у роботі рішення дають можливість автоматизувати конфігурування мікросервісів в умовах динамічного середовища, що дозволить зменшити вірогідність помилок, пов'язаних із людським фактором, та підвищити швидкість реакції системи на зміну умов.

Результати цієї роботи можуть бути використані для подальших досліджень застосування CBR-методів для інтелектуальної конфігурації мікросервісних застосунків. Перспективним є інтеграція реального моніторингу метрик із зовнішніх API для точнішого визначення поточного стану розподіленої системи.

Ще одним напрямком могла б бути оптимізація Cluster-Based Retrieval через адаптивну кількість кластерів, що залежить від щільності прецедентів.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Fowler, M., Lewis, J. Microservices: A Definition of This New Architectural Term. Стаття [Електронний ресурс] URL: <https://martinfowler.com/articles/microservices.html> (Дата останнього звернення: 31.05.2025)
2. Aamodt, A., Plaza, E. «Case-Based Reasoning: Foundational Issues, Methodological Variations, and System Approaches». AI Communications, 1994. 7(1), 39-59 с.
3. Newman, S. «Building Microservices: Designing Fine-Grained Systems». O'Reilly Media, 2015. 280 с.
4. Richardson, C. «Microservices Patterns: With Examples in Java». Manning Publications, 2018. 528 с.
5. Ramamoorthi, V. «AI-Enhanced Performance Optimization for Microservice-Based Systems». Journal of Advanced Computing Systems (JACS), 2024. 4(9), 1-7 с.
6. Kolodner, J. L. «Case-based reasoning». Morgan Kaufmann, 1993. 663 с.
7. Thiagarajan, G., Bist, V., Nayak, P. «AI-Driven Configuration Drift Detection in Cloud Environments». International Journal of Communication Networks and Information Security (IJCNIS), 2024. 16(5), 721-743 с.
8. Thönes, J. «Microservices». IEEE Software, 2015. 32(1), 116-116 с.
9. Kubernetes Authors. Documentation: ConfigMaps. Сайт [Електронний ресурс] URL: <https://kubernetes.io/docs/concepts/configuration/configmap/> (Дата останнього звернення: 21.05.2025)
10. Ramamoorthi, V. «Real-Time Adaptive Orchestration of AI Microservices in Dynamic Edge Computing». Journal of Advanced Computing Systems (JACS), 2023. 3(3), 1-9 с.
11. Resource Management with Deep Reinforcement Learning / Mao, H., Alizadeh, M., Menache, I., Kandula, S. // Proceedings of the 15th ACM Workshop on Hot Topics in Networks. 2016. 27-33 с.

12. HashiCorp. Consul Documentation. Сайт [Электронный ресурс] URL: <https://developer.hashicorp.com/consul/docs> (Дата останнього звернення: 05.06.2025)
13. Spring. Spring Cloud Config. Сайт [Электронный ресурс] URL: <https://spring.io/projects/spring-cloud-config> (Дата останнього звернення: 21.05.2025)
14. Amazon Web Services. AWS Config Documentation. Сайт [Электронный ресурс] URL: <https://docs.aws.amazon.com/config/> (Дата останнього звернення: 21.05.2025)
15. OWASP Foundation. OAuth 2.0 Cheat Sheet. Сайт [Электронный ресурс] URL: https://cheatsheetseries.owasp.org/cheatsheets/OAuth_2.0_Cheat_Sheet.html (Дата останнього звернення: 21.05.2025)
16. FastAPI. FastAPI Documentation. Сайт [Электронный ресурс] URL: <https://fastapi.tiangolo.com/> (Дата останнього звернення: 29.05.2025)
17. Docker Inc. Docker Documentation. Сайт [Электронный ресурс] URL: <https://docs.docker.com/> (Дата останнього звернення: 27.05.2025)
18. SQLAlchemy. SQLAlchemy Documentation. Сайт [Электронный ресурс] URL: <https://docs.sqlalchemy.org/> (Дата останнього звернення: 20.05.2025)
19. Awad, M., Khanna, R. «Efficient Learning Machines: Theories, Concepts, and Applications for Engineers and System Designers». Apress, 2015. 344 с.
20. Martin, R. C. «Clean Architecture: A Craftsman's Guide to Software Structure and Design». Prentice Hall, 2017. 432 с.
21. Fowler, M. «UML distilled: a brief guide to the standard object modeling language (3rd ed.)». Addison-Wesley Professional, 2004. 192 с.
22. Elmasri, R., Navathe, S. B. «Fundamentals of database systems (7th ed.)». Pearson, 2015. 1248 с.

23. scikit-learn developers. scikit-learn documentation. Сайт [Электронный ресурс] URL: <https://scikit-learn.org/> (Дата последнего обращения: 26.05.2025)
24. Passlib Team. Passlib Documentation. Сайт [Электронный ресурс] URL: <https://passlib.readthedocs.io/> (Дата последнего обращения: 26.05.2025)
25. aio-lib. slowapi documentation. Сайт [Электронный ресурс] URL: <https://slowapi.readthedocs.io/> (Дата последнего обращения: 23.05.2025)
26. aio-lib. aiohttp documentation. Сайт [Электронный ресурс] URL: <https://docs.aiohttp.org/> (Дата последнего обращения: 23.05.2025)
27. psutil authors. psutil documentation. Сайт [Электронный ресурс] URL: <https://psutil.readthedocs.io/> (Дата последнего обращения: 23.05.2025)
28. Han, J., Pei, J., Kamber, M. «Data Mining: Concepts and Techniques». Morgan Kaufmann, 2011. 744 с.
29. Richter, M. M., Weber, R. O. «Case-based reasoning: a textbook». Springer, 2013. 233 с.

ДОДАТОК А

Лістинг А.1 – Програмна реалізація методу Cluster-Based Retrieval.

```
def cluster_based_retrieval(db: Session, current_status: Dict,
n_clusters: int = 3) -> Dict:
    cases = db.query(Case).all()
    if not cases:
        logger.warning("No cases found in the database")
        return {}

    data = [
        [
            c.cpu_load,
            c.memory_usage,
            c.disk_usage,
            c.network_usage,
            c.performance_metric,
            c.response_time,
            c.operational_costs,
            c.availability
        ]
        for c in cases
    ]
    if not data:
        logger.warning("No valid data for clustering")
        return {}

    kmeans = KMeans(n_clusters=n_clusters, random_state=42)
    labels = kmeans.fit_predict(data)
    centroids = kmeans.cluster_centers_

    current_data = np.array([
        current_status.get("cpu_load", 0.0),
        current_status.get("memory_usage", 0.0),
        current_status.get("disk_usage", 0.0),
```

Продовження 1 Лістингу A.1

```

        current_status.get("network_usage", 0.0),
        current_status.get("performance_metric", 0.0),
        current_status.get("response_time", 0.0),
        current_status.get("operational_costs", 0.0),
        current_status.get("availability", 100.0)
    ]).reshape(1, -1)

    distances = np.linalg.norm(centroids - current_data, axis=1)
    nearest_cluster_idx = np.argmin(distances)

    cluster_cases = [
        {
            "cpu_load": c.cpu_load,
            "memory_usage": c.memory_usage,
            "disk_usage": c.disk_usage,
            "network_usage": c.network_usage,
            "performance_metric": c.performance_metric,
            "response_time": c.response_time,
            "operational_costs": c.operational_costs,
            "availability": c.availability,
            "configs": json.loads(c.configs)
        }
        for c, label in zip(cases, labels)
        if label == nearest_cluster_idx
    ]

    if not cluster_cases:
        logger.warning(f"No cases found in cluster {nearest_cluster_idx}")
        return {}

    best_case = min(cluster_cases, key=lambda x:
x["performance_metric"])

```

Продовження 2 Лістингу A.1

```

        logger.info(f"Best case selected:
configs={json.dumps(best_case['configs'], indent=2)}")
        return best_case["configs"]

```

Лістинг A.2 – Програмна реалізація методу Feature-Based Retrieval.

```

def calculate_feature_distance(case1: Dict, case2: Dict, features:
List[str]) -> float:

```

```

    distance = 0.0
    for metric in features:
        value1 = case1.get(metric, 0.0)
        value2 = case2.get(metric, 0.0)
        distance += (value1 - value2) ** 2
    return math.sqrt(distance)

```

```

def feature_based_retrieval(db: Session, current_status: Dict,
features: Optional[List[str]] = None, k: int = 3) -> Dict:

```

```

    if not features:
        features = ["cpu_load", "memory_usage",
"performance_metric"]

```

```

    valid_features = ["cpu_load", "memory_usage", "disk_usage",
"network_usage", "performance_metric", "response_time",
"operational_costs", "availability"]

```

```

    for feature in features:
        if feature not in valid_features:
            logger.error(f"Invalid feature: {feature}")
            raise ValueError(f"Invalid feature: {feature}. Must be
one of {valid_features}")

```

```

    cases = db.query(Case).all()
    if not cases:
        logger.warning("No cases found in the database")
        return {}

```

Продовження 1 Лістингу A.2

```

case_list = [
    {
        "cpu_load": c.cpu_load,
        "memory_usage": c.memory_usage,
        "disk_usage": c.disk_usage,
        "network_usage": c.network_usage,
        "performance_metric": c.performance_metric,
        "response_time": c.response_time,
        "operational_costs": c.operational_costs,
        "availability": c.availability,
        "configs": json.loads(c.configs)
    }
    for c in cases
]

distances = [
    (calculate_feature_distance(current_status, case,
features), case)
    for case in case_list
]

distances.sort(key=lambda x: x[0])
nearest_cases = [case for _, case in distances[:k]]

best_case = min(nearest_cases, key=lambda x:
x["performance_metric"])
logger.info(f"Best case selected:
performance_metric={best_case['performance_metric']}")
return best_case["configs"]

```

Лістинг A.3 – Програмна реалізація методу Indexing and Hashing.

```

class LSHIndex:
    def __init__(self, num_planes: int = 10, num_buckets: int =
100):

```

Продовження 1 Лістингу А.3

```

        self.num_planes = num_planes
        self.num_buckets = num_buckets
        self.planes = None
        self.buckets = [[] for _ in range(num_buckets)]

    def _generate_planes(self, dim: int):
        self.planes = np.random.randn(self.num_planes, dim)

    def _hash_vector(self, vector: np.ndarray) -> int:
        signs = np.dot(self.planes, vector) >= 0
        hash_code = sum(1 << i for i, sign in enumerate(signs) if
sign)
        return hash_code % self.num_buckets

    def index(self, vectors: List[np.ndarray], indices:
List[int]):
        if not vectors:
            return
        if self.planes is None:
            self._generate_planes(len(vectors[0]))
        for idx, vector in zip(indices, vectors):
            bucket_idx = self._hash_vector(vector)
            self.buckets[bucket_idx].append(idx)

    def query(self, query_vector: np.ndarray, k: int = 3) ->
List[int]:
        if self.planes is None:
            return []
        bucket_idx = self._hash_vector(query_vector)
        return self.buckets[bucket_idx][:k]

def indexing_hashing(db: Session, current_status: Dict, k: int =
3) -> Dict:
    logger.info("Starting indexing_hashing retrieval")

```

Продовження 2 Лістингу А.3

```

cases = db.query(Case).all()
if not cases:
    logger.warning("No cases found in the database")
    return {}

data = [
    [
        c.cpu_load,
        c.memory_usage,
        c.disk_usage,
        c.network_usage,
        c.performance_metric,
        c.response_time,
        c.operational_costs,
        c.availability
    ]
    for c in cases
]
if not data:
    logger.warning("No valid data for indexing")
    return {}

vectors = np.array(data)
indices = list(range(len(cases)))

lsh = LSHIndex(num_planes=10, num_buckets=100)
lsh.index(vectors, indices)

current_vector = np.array([
    current_status.get("cpu_load", 0.0),
    current_status.get("memory_usage", 0.0),
    current_status.get("disk_usage", 0.0),
    current_status.get("network_usage", 0.0),
    current_status.get("performance_metric", 0.0),

```

Продовження 3 Лістингу А.3

```

        current_status.get("response_time", 0.0),
        current_status.get("operational_costs", 0.0),
        current_status.get("availability", 100.0)
    ])

    nearest_indices = lsh.query(current_vector, k=k)
    if not nearest_indices:
        logger.warning("No similar cases found")
        return {}

    nearest_cases = [
        {
            "cpu_load": cases[idx].cpu_load,
            "memory_usage": cases[idx].memory_usage,
            "disk_usage": cases[idx].disk_usage,
            "network_usage": cases[idx].network_usage,
            "performance_metric": cases[idx].performance_metric,
            "response_time": cases[idx].response_time,
            "operational_costs": cases[idx].operational_costs,
            "availability": cases[idx].availability,
            "configs": json.loads(cases[idx].configs)
        }
        for idx in nearest_indices
    ]

    best_case = min(nearest_cases, key=lambda x:
x["performance_metric"])
    logger.info(f"Best case selected:
performance_metric={best_case['performance_metric']}")
    return best_case["configs"]

```

Лістинг А.4 – Програмна реалізація методів KNN.

```

def calculate_distance(case1: Dict, case2: Dict, weights:
Optional[Dict] = None) -> float:

```

Продовження 1 Лістингу A.4

```

    metrics = ["cpu_load", "memory_usage", "disk_usage",
               "network_usage", "performance_metric", "response_time",
               "operational_costs", "availability"]
    distance = 0.0
    for metric in metrics:
        value1 = case1.get(metric, 0.0)
        value2 = case2.get(metric, 0.0)
        if weights:
            weight = weights.get(metric, 1.0)
            distance += weight * (value1 - value2) ** 2
        else:
            distance += (value1 - value2) ** 2
    return math.sqrt(distance)

def knn_retrieval(db: Session, current_status: Dict, k: int = 3)
-> Dict:
    cases = db.query(Case).all()
    if not cases:
        logger.warning("No cases found in the database")
        return {}

    case_list = [
        {
            "cpu_load": c.cpu_load,
            "memory_usage": c.memory_usage,
            "disk_usage": c.disk_usage,
            "network_usage": c.network_usage,
            "performance_metric": c.performance_metric,
            "response_time": c.response_time,
            "operational_costs": c.operational_costs,
            "availability": c.availability,
            "configs": json.loads(c.configs)
        }
        for c in cases
    ]

```

Продовження 2 Лістингу A.4

```

    ]

    distances = [
        (calculate_distance(current_status, case), case)
        for case in case_list
    ]

    distances.sort(key=lambda x: x[0])
    nearest_cases = [case for _, case in distances[:k]]

    best_case = min(nearest_cases, key=lambda x:
x["performance_metric"])
    logger.info(f"Best case selected:
performance_metric={best_case['performance_metric']}")
    return best_case["configs"]

def weighted_knn_retrieval(db: Session, current_status: Dict,
weights: Dict, k: int = 3) -> Dict:
    cases = db.query(Case).all()
    if not cases:
        logger.warning("No cases found in the database")
        return {}

    case_list = [
        {
            "cpu_load": c.cpu_load,
            "memory_usage": c.memory_usage,
            "disk_usage": c.disk_usage,
            "network_usage": c.network_usage,
            "performance_metric": c.performance_metric,
            "response_time": c.response_time,
            "operational_costs": c.operational_costs,
            "availability": c.availability,
            "configs": json.loads(c.configs)
        }
    ]

```

Продовження 3 Лістингу A.4

```

        }
        for c in cases
    ]

    distances = [
        (calculate_distance(current_status, case, weights), case)
        for case in case_list
    ]

    distances.sort(key=lambda x: x[0])
    nearest_cases = [case for _, case in distances[:k]]

    best_case = min(nearest_cases, key=lambda x:
x["performance_metric"])
    logger.info(f"Best case selected:
performance_metric={best_case['performance_metric']}")
    return best_case["configs"]

```